Tell us about your PDF experience.

C# documentation

Learn how to write any application using the C# programming language.

Choose your path

GET STARTED

New to programming? Start with beginner tutorials

In-browser tutorial: Hello world

C# beginner video series

OVERVIEW

Experienced developer? A tour of C#

Learn C# for Java developers

Learn C# for JavaScript developers

Learn C# for Python developers

TRAINING

Foundational C# Certification

What you can build with C#

Fundamentals

OVERVIEW

A tour of C#

Inside a C# program

C# highlights video series

C# language strategy

CONCEPT

Type system

Object oriented programming

Functional techniques

Exceptions

Coding style

TUTORIAL

Build file-based apps

Intro to classes

Object oriented C#

Converting types

Pattern matching

Use LINQ to query data

What's new

WHAT'S NEW

What's new in C# 14

What's new in C# 13

What's new in C# 12

TUTORIAL

Explore record types

Explore top level statements

REFERENCE

Breaking changes in the C# compiler

Version compatibility

Key concepts

OVERVIEW

What you can build with C#

C# language strategy

Programming concepts

CONCEPT

Language Integrated Query (LINQ)

Asynchronous programming

TRAINING

Learn C# for Java developers

Learn C# for JavaScript developers

Learn C# for Python developers

C# language reference

REFERENCE

Language reference

Keywords

Operators and expressions

Tokens

Stay in touch

REFERENCE

C# on Q&A

.NET developer community

C# on Stack Overflow

C# on Discord

YouTube

A tour of the C# language

Summarize this article for me

This article gives you a high-level overview of the C# language. Whether you're writing your first program or you're an experienced developer exploring C#, you'll find the key concepts and features here.

> **Tip**
>
> **New to programming?** Skim this article for the big picture, then start the beginner tutorials to write code hands-on.
>
> **Coming from Java, JavaScript, or Python?** Read the tips for Java, JavaScript, or Python developers alongside this article.
>
> **Experienced developer new to C#?** This article covers what makes C# distinctive. See What you can build with C# to find the workload that fits your goals.

The C# language is the most popular language for the .NET platform, a free, cross-platform, open source development environment. C# programs can run on many different devices, from Internet of Things (IoT) devices to the cloud and everywhere in between. You can write apps for phone, desktop, and laptop computers and servers. See What you can build with C# for an overview of application types.

C# is a cross-platform general purpose language that makes developers productive while writing highly performant code. C# has broad support in the ecosystem and many workloads. Based on object-oriented principles, it incorporates many features from other paradigms, not least functional programming. Low-level features support high-efficiency scenarios without writing unsafe code. Most of the runtime and libraries are written in C#, and advances in C# often benefit all .NET developers.

C# is in the C family of languages. C# syntax is familiar if you used C, C++, JavaScript, TypeScript, or Java. Like C and C++, semicolons (`;`) define the end of statements. C# identifiers are case-sensitive. C# has the same use of braces, `{` and `}`, control statements like `if`, `else`, and `switch`, and looping constructs like `for` and `while`. C# also has a `foreach` statement for any collection type.

Hello world

The "Hello, World" program traditionally introduces a programming language. Here's the program in C#:

C#

```
// This line prints "Hello, World"
Console.WriteLine("Hello, World");
```

The line starting with `//` is a *single line comment*. C# single line comments start with `//` and continue to the end of the current line. C# also supports *multi-line comments*. Multi-line comments start with `/*` and end with `*/`. The `WriteLine` method of the `Console` class, which is in the `System` namespace, produces the output of the program. The standard class libraries provide this class, and every C# program automatically references these libraries by default. Another program form requires you to declare the containing class and method for the program's entry point. The compiler synthesizes these elements when you use top-level statements.

This alternative format is still valid and contains many of the basic concepts in all C# programs. Many existing C# samples use the following equivalent format:

C#

```
using System;
namespace TourOfCsharp;

class Program
{
    static void Main()
    {
        // This line prints "Hello, World"
        Console.WriteLine("Hello, World");
    }
}
```

The preceding "Hello, World" program starts with a `using` directive that references the `System` namespace. Namespaces provide a hierarchical means of organizing C# programs and libraries. Namespaces contain types and other namespaces. For example, the `System` namespace contains many types, such as the `Console` class referenced in the program, and many other namespaces, such as `IO` and `Collections`. A `using` directive that references a given namespace enables unqualified use of the types that are members of that namespace. Because of the `using` directive, the program can use `Console.WriteLine` as shorthand for `System.Console.WriteLine`. In the earlier example, that namespace was implicitly included.

> **Note**
>
> The preceding program declares the `Program` class with a single member: the method named `Main`. Many existing C# samples and tutorials use this format. By convention, when

there are no top-level statements, a static method named `Main` serves as the **entry point** of a C# program. Both formats compile to the same code. You don't need to use this format—file-based apps and top-level statements are simpler ways to get started.

File-based apps

C# is a *compiled* language. In most C# programs, you use the dotnet build command to compile a group of source files into a binary package. Then, you use the dotnet run command to run the program. (You can simplify this process because `dotnet run` compiles the program before running it if necessary.) These tools support a rich language of configuration options and command-line switches. The `dotnet` command line interface (CLI), which is included in the .NET SDK, provides many tools to generate and modify C# files.

Beginning with C# 14 and .NET 10, you can create *file-based apps*, which simplifies building and running C# programs. You use the `dotnet run` command to run a program contained in a single `*.cs` file. For example, if the following code is stored in a file named `hello-world.cs`, you can run it by typing `dotnet run hello-world.cs`:

C#

```
#!/usr/bin/env dotnet

Console.WriteLine("Hello, World!");
```

The first line of the program contains the `#!` sequence (shebang) for unix shells. The location of the `dotnet` CLI can vary on different distributions. On any unix system, if you set the *execute* (`+x`) permission on such a C# file that contains the shebang directive, you can run the C# file directly from the command line:

Bash

```
./hello-world.cs
```

The source for these programs must be a single file, but otherwise all C# syntax is valid. You can use file-based apps for small command-line utilities, prototypes, or other experiments.

Familiar C# features

C# is approachable for beginners yet offers advanced features for experienced developers writing specialized applications. You can be productive quickly. You can learn more specialized

techniques as you need them for your applications.

C# apps benefit from the runtime's automatic memory management. C# apps also use the extensive runtime libraries provided by the .NET SDK. Some components are platform independent, like file system libraries, data collections, and math libraries. Others are specific to a single workload, like the ASP.NET Core web libraries or the .NET MAUI UI library. A rich open source ecosystem on NuGet augments the libraries that are part of the runtime. These libraries provide even more components you can use.

C# is a *strongly typed* language. Every variable you declare has a type known at compile time. The compiler, or editing tools, tell you if you're using that type incorrectly. You can fix those errors before you ever run your program. Fundamental data types are built into the language and runtime: value types like `int`, `double`, `char`, reference types like `string`, arrays, and other collections. As you write your programs, you create your own types. Those types can be `struct` types for values, or `class` types that define object-oriented behavior. You can add the `record` modifier to either `struct` or `class` types so the compiler synthesizes code for equality comparisons. You can also create `interface` definitions, which define a contract, or a set of members, that a type implementing that interface must provide. You can also define generic types and methods. Generics use *type parameters* to provide a placeholder for an actual type when used.

As you write code, you define functions, also called methods, as members of `struct` and `class` types. These methods define the behavior of your types. You can overload methods with different numbers or types of parameters. Methods can optionally return a value. In addition to methods, C# types can have properties, which are data elements backed by functions called *accessors*. C# types can define events, which allow a type to notify subscribers of important actions. C# supports object oriented techniques such as inheritance and polymorphism for `class` types.

C# apps use exceptions to report and handle errors. This practice is familiar if you used C++ or Java. Your code throws an exception when it can't do what was intended. Other code, no matter how many levels up the call stack, can optionally recover by using a `try` - `catch` block.

> **Tip**
>
> To learn more about types, methods, and exceptions, visit the **C# fundamentals** section. It covers the **type system**, **object-oriented programming**, and **exception handling** in depth.

Distinctive C# features

Some elements of C# might be less familiar.

C# provides pattern matching. Those expressions enable you to inspect data and make decisions based on its characteristics. Pattern matching provides a great syntax for control flow based on data. The following code shows how methods for the boolean *and*, *or*, and *xor* operations could be expressed using pattern matching syntax:

C#

```
public static bool Or(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => true,
        (false, true) => true,
        (false, false) => false,
    };

public static bool And(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (true, false) => false,
        (false, true) => false,
        (false, false) => false,
    };
public static bool Xor(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => false,
        (true, false) => true,
        (false, true) => true,
        (false, false) => false,
    };
```

Pattern matching expressions can be simplified by using `_` as a catchall for any value. The following example shows how you can simplify the *and* method:

C#

```
public static bool ReducedAnd(bool left, bool right) =>
    (left, right) switch
    {
        (true, true) => true,
        (_, _) => false,
    };
```

The preceding examples also declare *tuples*, lightweight data structures. A *tuple* is an ordered, fixed-length sequence of values with optional names and individual types. You enclose the

sequence in `(` and `)` characters. The declaration `(left, right)` defines a tuple with two boolean values: `left` and `right`. Each switch arm declares a tuple value such as `(true, true)`. Tuples provide convenient syntax to declare a single value with multiple values of any type.

Collection expressions provide a common syntax to provide collection values. You write values or expressions between `[` and `]` characters and the compiler converts that expression to the required collection type:

C#

```
int[] numbers = [1, 2, 3, 4, 5, 6, 7, 8, 9, 10];
List<string> names = ["Alice", "Bob", "Charlie", "David"];

IEnumerable<int> moreNumbers = [.. numbers, 11, 12, 13];
IEnumerable<string> empty = [];
```

The previous example shows different collection types that you can initialize by using collection expressions. One example uses the `[]` empty collection expression to declare an empty collection. Another example uses the `..` *spread element* to expand a collection and add all its values to the collection expression.

You can use *index* and *range* expressions to retrieve one or more elements from an indexable collection:

C#

```
string second = names[1]; // 0-based index
string last = names[^1]; // ^1 is the last element
int[] smallNumbers = numbers[0..5]; // 0 to 4
```

The `^` index indicates *from the end* rather than from the start. The `^0` element is one past the end of the collection, so `^1` is the last element. The `..` in a range expression denotes the range of elements to include. The range starts with the first index and includes all elements up to, but not including, the element at the last index.

For more information about index and range expressions, see the Explore indexes and ranges article.

Language integrated query (LINQ) provides a common pattern-based syntax to query or transform any collection of data. LINQ unifies the syntax for querying in-memory collections, structured data like XML or JSON, database storage, and even cloud based data APIs. You learn one set of syntax and you can search and manipulate data regardless of its storage. The following query finds all students whose grade point average is greater than 3.5:

C#

```
var honorRoll = from student in Students
                where student.GPA > 3.5
                select student;
```

The preceding query works for many storage types represented by `Students`. It could be a collection of objects, a database table, a cloud storage blob, or an XML structure. The same query syntax works for all storage types.

The Task based asynchronous programming model enables you to write code that reads as though it runs synchronously, even though it runs asynchronously. It utilizes the `async` and `await` keywords to describe methods that are asynchronous, and when an expression evaluates asynchronously. The following sample awaits an asynchronous web request. When the asynchronous operation completes, the method returns the length of the response:

C#

```
public static async Task<int> GetPageLengthAsync(string endpoint)
{
    var client = new HttpClient();
    var uri = new Uri(endpoint);
    byte[] content = await client.GetByteArrayAsync(uri);
    return content.Length;
}
```

C# also supports an `await foreach` statement to iterate a collection backed by an asynchronous operation, like a GraphQL paging API. The following sample reads data in chunks, returning an iterator that provides access to each element when it's available:

C#

```
public static async IAsyncEnumerable<int> ReadSequence()
{
    int index = 0;
    while (index < 100)
    {
        int[] nextChunk = await GetNextChunk(index);
        if (nextChunk.Length == 0)
        {
            yield break;
        }
        foreach (var item in nextChunk)
        {
            yield return item;
        }
        index++;
```

```
    }
}
```

Callers can iterate the collection by using an `await foreach` statement:

C#

```
await foreach (var number in ReadSequence())
{
    Console.WriteLine(number);
}
```

Finally, as part of the .NET ecosystem, you can use Visual Studio, or Visual Studio Code with the C# DevKit. These tools provide rich understanding of C#, including the code you write. They also provide debugging capabilities.

> **Tip**
>
> To learn more about pattern matching, LINQ, and async programming, see the **functional techniques**, **LINQ overview**, and **asynchronous programming** sections.

Next steps

This article provided a quick tour of the C# language. Here's where to go next, depending on your experience:

- **Start coding**: Work through the beginner tutorials to learn C# step by step.
- **Go deeper**: Visit the C# fundamentals section for detailed coverage of the type system, object-oriented programming, and error handling.
- **Build something**: Explore what you can build with C# to find ideas that interest you.
- **Coming from another language?** Read the guides for Java, JavaScript, or Python developers.
- **Build file-based apps**: Learn how to build file-based apps for small programs and prototypes.

Last updated on 02/23/2026

What you can build with C#

Summarize this article for me

C# supports a wide range of application types. Whatever kind of software you want to build, there's likely a workload for it. This article gives you an overview of the most common types of applications, with links to get started on each one.

> **Tip**
>
> **New to programming?** Don't worry about choosing a workload yet. Focus on learning the C# language first. You can explore these application types once you're comfortable with the basics.
>
> **Experienced developer?** Jump directly to the workload that matches your goals. Each section links to the documentation and tutorials you need.

AI and machine learning

C# integrates with AI and machine learning tools:

- **Agent Framework** - Build agents and workflows for Azure, OpenAI, Anthropic, Ollama, and more.
- **Azure AI services** - Access pre-built AI capabilities like vision, language understanding, and speech recognition.
- **ML.NET** - Build custom machine learning models in C# without requiring deep ML expertise.

Get started: ML.NET tutorials.

Web applications

Build web applications with ASP.NET Core, the cross-platform framework for creating modern web apps and APIs. You can build:

- **Server-rendered web apps** by using Razor Pages or MVC.
- **Interactive web UIs** by using Blazor, which lets you write client-side logic in C# instead of JavaScript.
- **Web APIs** and minimal APIs for backend services.

Get started: ASP.NET Core tutorials

Desktop applications

Use C# to build Windows desktop apps, and cross-platform desktop apps that run on Windows and macOS:

- **.NET MAUI** creates cross-platform desktop apps that run on Windows, macOS, Android, and iOS from a single codebase.
- **Windows Presentation Foundation (WPF)** builds rich Windows-only desktop apps with advanced graphics and data binding.
- **Windows Forms** provides a straightforward way to create Windows-only desktop apps with a visual designer.

Get started: .NET MAUI tutorials.

Mobile applications

Build native mobile apps for iOS and Android by using .NET MAUI. .NET MAUI lets you share code across mobile and desktop platforms while still accessing native device APIs for cameras, sensors, GPS, and more.

Get started: Build your first .NET MAUI app.

Cloud and microservices

C# is well suited for building cloud-native applications and microservices:

- **Azure SDK for .NET** provides libraries for working with Azure services like storage, messaging, and databases.
- **Worker Services** let you build long-running background services that run in the cloud or on-premises.
- **Aspire** streamlines building, running, deploying, debugging, and deploying distributed apps.

Get started: Get started with Azure and .NET.

Games

C# is one of the most popular languages for game development:

- **Unity** - The most widely used game engine for 2D and 3D games, uses C# as its scripting language.
- **MonoGame** - An open-source framework for creating cross-platform games.

- **CryEngine** - Supports C# for game scripting.

You can also use Visual Studio for game development with these engines.

Internet of Things (IoT)

Control devices and read sensor data by using the IoT libraries. You can run C# apps on devices like Raspberry Pi and other single-board computers to build IoT solutions.

Get started: IoT tutorials

Next steps

Now that you know what you can build, choose your next step:

- **Learn the language**: Start with the beginner C# tutorials to learn C# fundamentals.
- **Explore C#**: Read A tour of C# for an overview of key language features.
- **Dive into fundamentals**: Visit the C# fundamentals section for deeper coverage of the type system, object-oriented programming, and more.
- **Coming from another language?** See the roadmaps for Java, JavaScript/TypeScript, or Python developers.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 02/23/2026

Introduction to C#

Summarize this article for me

Welcome to the introduction to C# tutorials. These lessons start with interactive code that you can run in GitHub Codespaces. You can learn the basics of C# from the C# for Beginners video series before starting these interactive lessons.

> **Tip**
>
> **New to programming?** Start with tutorial 1 (**Hello world**) and work through each lesson in order.
>
> **Coming from another language?** You can skim tutorials 1–2 and start with **Tuples and types** for concepts that might be new.

All tutorials use file-based apps. You can use our GitHub Codespace, or install the .NET SDK, and you're ready to start.

https://www.youtube-nocookie.com/embed/9THmGiSPjBQ?si=3kUKFtOMLpEzeq7J

The first lessons explain C# concepts by using small snippets of code. You learn the basics of C# syntax and how to work with data types like strings, numbers, and Booleans. It's all interactive, and you'll be writing and running code within minutes. These first lessons assume no prior knowledge of programming or the C# language. Each lesson builds on the prior lessons. You should do them in order. However, if you have some programming experience, you can skip or skim the first lessons and start with any new concepts.

To use GitHub Codespaces, you need to create a free GitHub account.

Hello world

In the Hello world tutorial, you create the most basic C# program. You explore the `string` type and how to work with text.

Numbers in C#

In the Numbers in C# tutorial, you learn how computers store numbers and how to perform calculations with different numeric types. You learn the basics of rounding, and how to perform mathematical calculations using C#.

Tuples and types

In the Tuples and types tutorial, you learn to create types in C#. You can create *tuples*, *records*, *struct*, and *class* types. The capabilities of these different kinds of types reflect their different uses.

Branches and loops

The Branches and loops tutorial teaches the basics of selecting different paths of code execution based on the values stored in variables. You learn the basics of control flow, which is the basis of how programs make decisions and choose different actions.

List collection

The List collection lesson gives you a tour of the List collection type that stores sequences of data. You learn how to add and remove items, search for items, and sort the lists. You explore different kinds of lists.

Pattern matching

The Pattern matching lesson provides an introduction to *pattern matching*. Pattern matching enables you to compare an expression against a pattern. The success of the match determines which program logic to follow. Patterns can compare types, properties of a type, or contents of a list. You can combine multiple patterns by using `and`, `or`, and `not` logic. Patterns provide a rich vocabulary to inspect data and make decisions in your program based on that inspection.

What's next?

After you finish these tutorials, continue your learning journey:

- Build file-based apps: Learn how to create and run single-file C# programs from the command line.
- What you can build with C#: Explore the types of applications you can create with C#.
- C# type system: Dive deeper into value types, reference types, generics, and more.
- C# fundamentals: Learn about program structure, object-oriented programming, and error handling.

Last updated on 02/23/2026

Tutorial: Explore the C# language

Summarize this article for me

This tutorial teaches you C#. You write your first C# program and see the results of compiling and running your code. It contains a series of lessons that begin with a "Hello World" program. These lessons teach you the fundamentals of the C# language.

> **Tip**
>
> **New to programming?** Start here - this tutorial assumes no prior experience. **Coming from another language?** You might prefer to skim the code samples and jump ahead to Numbers in C# or Branches and loops.

In this tutorial, you:

- ✓ Launch a GitHub Codespace with a C# development environment.
- ✓ Create your first C# app.
- ✓ Create and use variables to store text data.
- ✓ Use runtime APIs with text data.

Prerequisites

You must have one of the following options:

- A GitHub account to use GitHub Codespaces. If you don't already have one, you can create a free account at GitHub.com.
- A computer with the following tools installed:
  - The .NET 10 SDK.
  - Visual Studio Code.
  - The C# DevKit.

Open Codespaces

To start a GitHub Codespace with the tutorial environment, open a browser window to the tutorial codespace repository. Select the green *Code* button, and the *Codespaces* tab. Then select the `+` sign to create a new Codespace using this environment.

Run your first program

1. When your codespace loads, create a new file in the *tutorials* folder named *hello-world.cs*.
2. Open your new file.
3. Type or copy the following code into *hello-world.cs*:

   C#

   ```
   Console.WriteLine("Hello, World!");
   ```

4. In the integrated terminal window, make the *tutorials* folder the current folder, and run your program:

   .NET CLI

   ```
   cd tutorials
   dotnet hello-world.cs
   ```

You ran your first C# program. It's a simple program that prints the message "Hello World!" It uses the Console.WriteLine method to print that message. `Console` is a type that represents the console window. `WriteLine` is a method of the `Console` type that prints a line of text to that text console.

Let's move on and explore more. The rest of this lesson explores working with the `string` type, which represents text in C#. Like the `Console` type, the `string` type has methods. The `string` methods work with text.

Declare and use variables

Your first program prints the `string` "Hello World!" on the screen.

> **Tip**
>
> As you explore C# (or any programming language), you make mistakes when you write code. The **compiler** finds those errors and reports them to you. When the output contains error messages, look closely at the example code and the code in your `.cs` file to see what to fix. That exercise helps you learn the structure of C# code. You can also ask Copilot to find differences or spot mistakes.

Your first program is limited to printing one message. You can write more useful programs by using *variables*. A *variable* is a symbol you can use to run the same code with different values. Let's try it!

1. Start with the following code:

   ```C#
   string aFriend = "Bill";
   Console.WriteLine(aFriend);
   ```

   The first line declares a variable, `aFriend`, and assigns it a value, "Bill". The second line prints the name.

2. You can assign different values to any variable you declare. You can change the name to one of your friends. Add these two lines following the code you already added. Make sure you keep the declaration of the `aFriend` variable and its initial assignment.

   > **ⓘ Important**
   >
   > Don't delete the declaration of `aFriend`.

3. Add the following code at the end of the preceding code:

   ```C#
   aFriend = "Maira";
   Console.WriteLine(aFriend);
   ```

   Notice that the same line of code prints two different messages, based on the value stored in the `aFriend` variable. You might notice that the word "Hello" is missing in the last two messages. Let's fix that now.

4. Modify the lines that print the message to the following code:

   ```C#
   Console.WriteLine("Hello " + aFriend);
   ```

5. Run the app again by using `dotnet hello-world.cs` to see the results.

   You've been using `+` to build strings from **variables** and **constant** strings. There's a better way. You can place a variable between `{` and `}` characters to tell C# to replace that text with the value of the variable. This process is called String interpolation.

6. If you add a `$` before the opening quote of the string, you can then include variables, like `aFriend`, inside the string between curly braces. Give it a try:

C#

```
Console.WriteLine($"Hello {aFriend}");
```

7. Run the app again by using `dotnet hello-world.cs` to see the results. Instead of "Hello {aFriend}", the message should be "Hello Maira".

Work with strings

Your last edit was your first look at what you can do with strings. Let's explore more.

You're not limited to a single variable between the curly braces.

1. Try the following code at the bottom of your app:

   C#

   ```
   string firstFriend = "Maria";
   string secondFriend = "Sage";
   Console.WriteLine($"My friends are {firstFriend} and {secondFriend}");
   ```

   Strings are more than a collection of letters. You can find the length of a string by using `Length`. `Length` is a **property** of a string and it returns the number of characters in that string.

2. Add the following code at the bottom of your app:

   C#

   ```
   Console.WriteLine($"The name {firstFriend} has {firstFriend.Length}
   letters.");
   Console.WriteLine($"The name {secondFriend} has {secondFriend.Length}
   letters.");
   ```

> **Tip**
>
> Now is a good time to explore on your own. You learned that `Console.WriteLine()` writes text to the screen. You learned how to declare variables and concatenate strings together. Experiment in your code. Your editor has a feature called *IntelliSense* that makes suggestions for what you can do. Type a `.` after the `d` in `firstFriend`. You see a list of suggestions for properties and methods you can use.

You've been using a *method*, Console.WriteLine, to print messages. A *method* is a block of code that implements some action. It has a name, so you can access it.

> **Tip**
>
> **Learn more:** Explore strings in depth, or read about methods and program structure in the C# Fundamentals section.

Remove whitespace from strings

Suppose your strings have leading or trailing spaces that you don't want to display. You want to **trim** the spaces from the strings. The Trim method and related methods TrimStart and TrimEnd perform this task. Use these methods to remove leading and trailing spaces.

1. Try the following code:

```C#
string greeting = "      Hello World!       ";
Console.WriteLine($"[{greeting}]");

string trimmedGreeting = greeting.TrimStart();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.TrimEnd();
Console.WriteLine($"[{trimmedGreeting}]");

trimmedGreeting = greeting.Trim();
Console.WriteLine($"[{trimmedGreeting}]");
```

The square brackets `[` and `]` help you visualize what the `Trim`, `TrimStart`, and `TrimEnd` methods do. The brackets show where whitespace starts and ends.

This sample reinforces a couple of important concepts for working with strings. The methods that manipulate strings return new string objects rather than making modifications in place. You can see that each call to any of the `Trim` methods returns a new string but doesn't change the original message.

Search and replace text in strings

You can use other methods to work with a string. For example, you might use a search and replace command in an editor or word processor. The Replace method does something similar in a string. It searches for a substring and replaces it with different text. The Replace method

takes two **parameters**. These parameters are the strings between the parentheses. The first string is the text to search for. The second string is the text to replace it with. Try it for yourself.

1. Add this code. Type it in to see the hints as you start typing `.Re` after the `sayHello` variable:

   ```
   C#
   ```

   ```
   string sayHello = "Hello World!";
   Console.WriteLine(sayHello);
   sayHello = sayHello.Replace("Hello", "Greetings");
   Console.WriteLine(sayHello);
   ```

   Two other useful methods make a string all caps or all lowercase. Try the following code.

2. Type it in to see how **IntelliSense** provides hints as you start to type `To`:

   ```
   C#
   ```

   ```
   Console.WriteLine(sayHello.ToUpper());
   Console.WriteLine(sayHello.ToLower());
   ```

   The other part of a *search and replace* operation is to find text in a string. You can use the Contains method for searching. It tells you if a string contains a substring inside it.

3. Try the following code to explore Contains:

   ```
   C#
   ```

   ```
   string songLyrics = "You say goodbye, and I say hello";
   Console.WriteLine(songLyrics.Contains("goodbye"));
   Console.WriteLine(songLyrics.Contains("greetings"));
   ```

   The Contains method returns a *boolean* value that tells you if the string you were searching for was found. A *boolean* stores either a `true` or a `false` value. When displayed as text output, they're capitalized: `True` and `False`, respectively. You learn more about *boolean* values in a later lesson.

Challenge

Two similar methods, StartsWith and EndsWith, also search for substrings in a string. These methods find a substring at the beginning or at the end of the string. Try to modify the previous sample to use StartsWith and EndsWith instead of Contains. Search for "You" or "goodbye" at the beginning of a string. Search for "hello" or "goodbye" at the end of a string.

> **Note**
>
> Watch your punctuation when you test for the text at the end of the string. If the string ends with a period, you must check for a string that ends with a period.

You should get `true` for starting with "You" and ending with "hello" and `false` for starting with or ending with "goodbye".

Did you come up with something like the following code (expand to see the answer):

▼ Details

C#

```
string songLyrics = "You say goodbye, and I say hello";
Console.WriteLine(songLyrics.StartsWith("You"));
Console.WriteLine(songLyrics.StartsWith("goodbye"));

Console.WriteLine(songLyrics.EndsWith("hello"));
Console.WriteLine(songLyrics.EndsWith("goodbye"));
```

For further reading on the `string` type:

- C# programming guide article on strings.
- How to tips on working with strings.

Cleanup resources

GitHub automatically deletes your Codespace after 30 days of inactivity. If you plan to explore more tutorials in this series, you can leave your Codespace provisioned. If you're ready to visit the .NET site to download the .NET SDK, you can delete your Codespace. To delete your Codespace, open a browser window and navigate to your Codespaces. You should see a list of your codespaces in the window. Select the three dots (`...`) in the entry for the learn tutorial codespace and select "delete".

Next steps

Continue to the next tutorial in this series, or explore related topics in C# Fundamentals:

Explore numbers in C#

- Strings — Learn more about the `string` type you used in this tutorial.
- Methods and program structure — Understand how C# programs are organized.

- File-based apps — Learn about the `dotnet run` command you used to run your code.

Last updated on 02/23/2026

Tutorial: How to use integer and floating point numbers in C#

Summarize this article for me

This tutorial teaches you about the numeric types in C#. You write small amounts of code, then you compile and run that code. The tutorial contains a series of lessons that explore numbers and math operations in C#. These lessons teach you the fundamentals of the C# language.

> **Tip**
>
> **New to programming?** Work through each section in order. **Coming from another language?** You might already know integer vs. floating-point distinctions - skim the basics and focus on the precision and limits and decimal type sections.

In this tutorial, you:

- ✓ Launch a GitHub Codespace with a C# development environment.
- ✓ Explore integer math.
- ✓ Learn order of operations.
- ✓ Learn integer limits and precision.
- ✓ Learn floating point types.
- ✓ Learn the decimal type.

Prerequisites

You must have one of the following options:

- A GitHub account to use GitHub Codespaces. If you don't already have one, you can create a free account at GitHub.com.
- A computer with the following tools installed:
  - The .NET 10 SDK.
  - Visual Studio Code.
  - The C# DevKit.

Explore integer math

To start a GitHub Codespace with the tutorial environment, open a browser window to the tutorial codespace repository. Select the green *Code* button, and the *Codespaces* tab. Then

select the `+` sign to create a new Codespace using this environment. If you completed the hello world tutorial, you can open that codespace instead of creating a new one.

1. When your codespace loads, create a new file in the *tutorials* folder named *numbers.cs*.
2. Open your new file.
3. Type or copy the following code into *numbers.cs*:

   C#

   ```
   int a = 18;
   int b = 6;
   int c = a + b;
   Console.WriteLine(c);
   ```

4. Run this code by typing the following commands in the integrated terminal:

   .NET CLI

   ```
   cd ./tutorials
   dotnet numbers.cs
   ```

   You saw one of the fundamental math operations with integers. The `int` type represents an **integer**, a zero, positive, or negative whole number. You use the `+` symbol for addition. Other common mathematical operations for integers include:

   - `-` for subtraction
   - `*` for multiplication
   - `/` for division

5. Start by exploring those different operations. Add these lines after the line that writes the value of `c`:

   C#

   ```
   // subtraction
   c = a - b;
   Console.WriteLine(c);

   // multiplication
   c = a * b;
   Console.WriteLine(c);

   // division
   ```

```
c = a / b;
Console.WriteLine(c);
```

6. Run this code by typing `dotnet numbers.cs` in the terminal window.

You can also experiment by writing multiple mathematics operations in the same line, if you'd like. Try `c = a + b - 12 * 17;` for example. Mixing variables and constant numbers is allowed.

> **Tip**
>
> As you explore C# (or any programming language), you might make mistakes when you write code. The **compiler** finds those errors and reports them to you. When the output contains error messages, look closely at the example code and the code in your window to see what to fix. You can also ask Copilot to find differences or spot mistakes. That exercise helps you learn the structure of C# code.

You finished the first step. Before you start the next section, move the current code into a separate *method*. A method is a series of statements grouped together and given a name. You call a method by writing the method's name followed by `()`. Organizing your code into methods makes it easier to start working with a new example.

> **Tip**
>
> **Learn more:** Read about methods and program structure in C# Fundamentals.

When you finish, your code should look like this:

C#

```
WorkWithIntegers();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
```

```
    Console.WriteLine(c);

    // division
    c = a / b;
    Console.WriteLine(c);
}
```

Explore order of operations

1. Comment out the call to `WorkingWithIntegers()`. This change makes the output less cluttered as you work in this section:

   C#

   ```
   //WorkWithIntegers();
   ```

   The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but not execute as code. The compiler doesn't generate any executable code from comments. Because `WorkWithIntegers()` is a method, you need to comment out only one line.

2. The C# language defines the precedence of different mathematics operations with rules consistent with the rules you learned in mathematics. Multiplication and division take precedence over addition and subtraction. Explore that precedence by adding the following code after the call to `WorkWithIntegers()`, and typing `dotnet numbers.cs` in the terminal window:

   C#

   ```
   int a = 5;
   int b = 4;
   int c = 2;
   int d = a + b * c;
   Console.WriteLine(d);
   ```

   The output demonstrates that the multiplication is performed before the addition.

3. You can force a different order of operation by adding parentheses around the operation or operations you want performed first. Add the following lines and run the app again:

   C#

   ```
   d = (a + b) * c;
   Console.WriteLine(d);
   ```

4. Explore more by combining many different operations. Add something like the following lines. Try `dotnet numbers` again in the terminal window.

   C#

   ```
   d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
   Console.WriteLine(d);
   ```

   You might notice an interesting behavior for integers. Integer division always produces an integer result, even when you'd expect the result to include a decimal or fractional portion.

5. If you didn't see this behavior, try the following code:

   C#

   ```
   int e = 7;
   int f = 4;
   int g = 3;
   int h = (e + f) / g;
   Console.WriteLine(h);
   ```

6. Type `dotnet numbers.cs` again in the terminal window to see the results.

Before moving on, take all the code you wrote in this section and put it in a new method. Call that new method `OrderPrecedence`. Your code should look something like this:

C#

```
// WorkWithIntegers();
OrderPrecedence();

void WorkWithIntegers()
{
    int a = 18;
    int b = 6;
    int c = a + b;
    Console.WriteLine(c);


    // subtraction
    c = a - b;
    Console.WriteLine(c);

    // multiplication
    c = a * b;
    Console.WriteLine(c);

    // division
```

```
    c = a / b;
    Console.WriteLine(c);
}

void OrderPrecedence()
{
    int a = 5;
    int b = 4;
    int c = 2;
    int d = a + b * c;
    Console.WriteLine(d);

    d = (a + b) * c;
    Console.WriteLine(d);

    d = (a + b) - 6 * c + (12 * 4) / 3 + 12;
    Console.WriteLine(d);

    int e = 7;
    int f = 4;
    int g = 3;
    int h = (e + f) / g;
    Console.WriteLine(h);
}
```

Explore integer precision and limits

The previous sample showed that integer division truncates the result. You can get the **remainder** by using the **remainder** operator, the `%` character.

1. Try the following code after the method call to `OrderPrecedence()`:

   C#

   ```
   int a = 7;
   int b = 4;
   int c = 3;
   int d = (a + b) / c;
   int e = (a + b) % c;
   Console.WriteLine($"quotient: {d}");
   Console.WriteLine($"remainder: {e}");
   ```

2. The C# integer type differs from mathematical integers in one other way: the `int` type has minimum and maximum limits. Try the following code to see those limits:

   C#

   ```
   int max = int.MaxValue;
   int min = int.MinValue;
   ```

```
Console.WriteLine($"The range of integers is {min} to {max}");
```

3. If a calculation produces a value that exceeds those limits, you get an **underflow** or **overflow** condition. The answer appears to wrap from one limit to the other. To see an example, add these two lines to your code:

   C#

   ```
   int what = max + 3;
   Console.WriteLine($"An example of overflow: {what}");
   ```

Notice that the answer is very close to the minimum (negative) integer. It's the same as `min + 2`. The addition operation **overflowed** the allowed values for integers. The answer is a large negative number because an overflow "wraps around" from the largest possible integer value to the smallest.

If the `int` type doesn't meet your needs, use other numeric types with different limits and precision. Let's explore those types of numbers next.

Work with the double type

The `double` numeric type represents a double-precision floating point number. Those terms might be new to you. A **floating point** number is useful for representing non-integral numbers that can be very large or small in magnitude. **Double-precision** is a relative term that describes the number of binary digits used to store the value. **Double precision** numbers have twice the number of binary digits as **single-precision**. On modern computers, you more commonly use double precision than single precision numbers. **Single precision** numbers are declared by using the `float` keyword. Let's explore.

1. Add the following code and see the result:

   C#

   ```
   double a = 5;
   double b = 4;
   double c = 2;
   double d = (a + b) / c;
   Console.WriteLine(d);
   ```

   Notice that the answer includes the decimal portion of the quotient.

2. Try a slightly more complicated expression with doubles. You can use the following values, or substitute other numbers:

```C#
double a = 19;
double b = 23;
double c = 8;
double d = (a + b) / c;
Console.WriteLine(d);
```

3. The range of a double value is much greater than integer values. Try the following code that you add to what you wrote so far:

```C#
double max = double.MaxValue;
double min = double.MinValue;
Console.WriteLine($"The range of double is {min} to {max}");
```

These values are printed in scientific notation. The number to the left of the `E` is the significand. The number to the right is the exponent, as a power of 10.

4. Just like decimal numbers in math, doubles in C# can have rounding errors. Try this code:

```C#
double third = 1.0 / 3.0;
Console.WriteLine(third);
```

You know that `0.3` is `3/10` and not exactly the same as `1/3`. Similarly, `0.33` is `33/100`. That value is closer to `1/3`, but still not exact. No matter how many decimal places you add, a rounding error remains.

Challenge

Try other calculations with large numbers, small numbers, multiplication, and division by using the `double` type. Try more complicated calculations. After you spend some time with the challenge, take the code you wrote and place it in a new method. Name that new method `WorkWithDoubles`.

Work with decimal types

You saw the basic numeric types in C#: integers and doubles. There's one other type to learn: the `decimal` type. The `decimal` type has a smaller range but greater precision than `double`.

1. Let's take a look:

C#

```
decimal min = decimal.MinValue;
decimal max = decimal.MaxValue;
Console.WriteLine($"The range of the decimal type is {min} to {max}");
```

Notice that the range is smaller than the `double` type.

2. You can see the greater precision with the decimal type by trying the following code:

C#

```
double a = 1.0;
double b = 3.0;
Console.WriteLine(a / b);

decimal c = 1.0M;
decimal d = 3.0M;
Console.WriteLine(c / d);
```

Notice that the math with the decimal type has more digits to the right of the decimal point.

The `M` suffix on the numbers indicates that a constant should use the `decimal` type. Otherwise, the compiler assumes the `double` type.

> ⓘ **Note**
>
> The letter `M` is the most visually distinct letter between the `double` and `decimal` keywords.

Challenge

Now that you know the different numeric types, write code that calculates the area of a circle whose radius is 2.50 centimeters. Remember that the area of a circle is the radius squared multiplied by PI. One hint: the runtime contains a constant for PI, Math.PI that you can use for that value. Math.PI, like all constants declared in the `System.Math` namespace, is a `double` value. For that reason, you should use `double` instead of `decimal` values for this challenge.

You should get an answer between 19 and 20.

▼ Details

C#

```
double radius = 2.50;
double area = Math.PI * radius * radius;
```

```
Console.WriteLine(area);
```

Try some other formulas if you'd like.

Cleanup resources

GitHub automatically deletes your Codespace after 30 days of inactivity. If you plan to explore more tutorials in this series, you can leave your Codespace provisioned. If you're ready to visit the .NET site to download the .NET SDK, you can delete your Codespace. To delete your Codespace, open a browser window and go to your Codespaces. You should see a list of your codespaces in the window. Select the three dots (`...`) in the entry for the learn tutorial codespace and select **delete**.

Next steps

Continue with the next tutorial in this series:

Tuples and types

Or explore related topics in C# Fundamentals:

- The C# type system — Dive deeper into the types you used in this tutorial.
- Methods and program structure — Learn more about the methods you created to organize your code.
- What you can build with C# — See the kinds of apps you can create with what you're learning.
- Integral numeric types
- Floating-point numeric types
- Built-in numeric conversions

Last updated on 02/23/2026

Tutorial: Create types in C#

Summarize this article for me

This tutorial teaches you how to create types in C#. You write small amounts of code, then you compile and run that code. The tutorial contains a series of lessons that explore different kinds of types in C#. These lessons teach you the fundamentals of the C# language.

> **Tip**
>
> **New to programming?** Work through each section in order - tuples, records, and then structs and classes. **Coming from another language?** If you already know classes and structs, focus on **tuples** and **record types**, which might be new to you.

The preceding tutorials worked with text and numbers. Strings and numbers are *simple types*: They each store one single value. As your programs grow larger, you need to work with more sophisticated data structures. C# provides different kinds of types you can define when you need data structures with more fields, properties, or behavior. Let's start to explore those types.

In this tutorial, you:

- ✓ Create and manipulate tuple types.
- ✓ Create record types.
- ✓ Learn about struct, class, and interface types.

Prerequisites

You must have one of the following options:

- A GitHub account to use GitHub Codespaces. If you don't already have one, you can create a free account at GitHub.com.
- A computer with the following tools installed:
  - The .NET 10 SDK.
  - Visual Studio Code.
  - The C# DevKit.

To use codespaces, you need a GitHub account. If you don't already have one, you can create a free account at GitHub.com.

Tuples

To start a GitHub Codespace with the tutorial environment, open a browser window to the tutorial codespace repository. Select the green *Code* button, and the *Codespaces* tab. Then select the `+` sign to create a new Codespace using this environment. If you completed other tutorials in this series, you can open that codespace instead of creating a new one.

1. When your codespace loads, create a new file in the *tutorials* folder named *tuples.cs*.

2. Open your new file.

3. Type or copy the following code into *tuples.cs*:

   C#

   ```
   var pt = (X: 1, Y: 2);

   var slope = (double)pt.Y / (double)pt.X;
   Console.WriteLine($"A line from the origin to the point {pt} has a slope of
   {slope}.");
   ```

4. Run your program by typing the following commands in the integrated terminal window:

   .NET CLI

   ```
   cd tutorials
   dotnet tuples.cs
   ```

   Tuples are an ordered sequence of values with a fixed length. Each element of a tuple has a type and an optional name.

   > **Tip**
   >
   > As you explore C# (or any programming language), you make mistakes when you write code. The **compiler** finds those errors and reports them to you. When the output contains error messages, look closely at the example code and your code to see what to fix. You can also ask Copilot to find differences or spot any mistakes. That exercise helps you learn the structure of C# code.

5. Add the following code after the previous code to modify a tuple member:

   C#

   ```
   pt.X = pt.X + 5;
   Console.WriteLine($"The point is now at {pt}.");
   ```

6. You can also create a new tuple that's a modified copy of the original using a `with` expression. Add the following code after the existing code and type `dotnet tuples.cs` in the terminal window to see the results:

   ```
   C#

   var pt2 = pt with { Y = 10 };
   Console.WriteLine($"The point 'pt2' is at {pt2}.");
   ```

   The tuple `pt2` contains the `X` value of `pt` (6), and `pt2.Y` is 10. Tuples are structural types. In other words, tuple types don't have names like `string` or `int`. A tuple type is defined by the number of members, referred to as *arity*, and the types of those members. The member names are for convenience. You can assign a tuple to a tuple with the same arity and types even if the members have different names.

7. You can add the following code after the code you already wrote:

   ```
   C#

   var subscript = (A: 0, B: 0);
   subscript = pt;
   Console.WriteLine(subscript);
   ```

8. Try it by typing `dotnet tuples.cs` in the terminal window. The variable `subscript` has two members, both of which are integers. Both `subscript` and `pt` represent instances of the same tuple type: a tuple containing two `int` members.

   Tuples are easy to create: You declare multiple members enclosed in parentheses. All the following declarations define different tuples with different arities and member types.

9. Add the following code to create new tuple types:

   ```
   C#

   var namedData = (Name: "Morning observation", Temp: 17, Wind: 4);
   var person = (FirstName: "", LastName: "");
   var order = (Product: "guitar picks", style: "triangle", quantity: 500,
   UnitPrice: 0.10m);
   ```

10. Try this change by typing `dotnet tuples.cs` again in the terminal window.

While tuples are easy to create, they're limited in their capabilities. Tuple types don't have names, so you can't convey meaning to the set of values. Tuple types can't add behavior. C# has other kinds of types you can create when your type defines behavior.

> **Tip**
>
> **Learn more:** Explore tuples and other type choices in C# Fundamentals.

Create record types

Tuples work well when you want multiple values in the same structure. They're lightweight, and you can declare them as you use them. As your program grows, you might find that you use the same tuple type throughout your code. If your app works in the 2D graph space, the tuples that represent points might be common. When you find this pattern, declare a `record` type that stores those values and provides more capabilities.

1. Add the following code to declare and use a `record` type to represent a `Point`:

   C#

   ```
   public record Point(int X, int Y);
   ```

   Place the preceding code at the bottom of your source file. Type declarations like `record` declarations must follow executable statements in a file-based app.

2. Add the following code preceding the `record` declaration:

   C#

   ```
   Point pt3 = new Point(1, 1);
   var pt4 = pt3 with { Y = 10 };
   Console.WriteLine($"The two points are {pt3} and {pt4}");
   ```

   The `record` declaration is a single line of code for the `Point` type that stores the values `X` and `Y` in readonly properties. You use the name `Point` wherever you use that type. Properly named types, like `Point`, provide information about how the type is used. The additional code shows how to use a `with` expression to create a new point that's a modified copy of the existing point. The line `pt4 = pt3 with { Y = 10 }` says "`pt4` has the same values as `pt3` except that `Y` is assigned to 10." You can add any number of properties to change in a single `with` expression.

   The preceding `record` declaration is a single line of code that ends in `;`. You can add behavior to a `record` type by declaring *members*. A record member can be a function or more data elements. The members of a type are in the type declaration, between `{` and `}` characters.

3. Delete the `;` and add the following lines of code after the `record` declaration:

```C#
{
    public double Slope() => (double)Y / (double)X;
}
```

4. Add the following code before the `record` declaration, after the line containing the `with` expression:

```C#
double slopeResult = pt4.Slope();
Console.WriteLine($"The slope of {pt4} is {slopeResult}");
```

5. Type `dotnet tuples.cs` in the terminal window to run this version.

   You added formality to the *tuple* representing an `X` and `Y` value. You made it a `record` that defined a named type and included a member to calculate the slope. A `record` type is a shorthand for a `record class`: A `class` type that includes extra behavior.

6. You can modify the `Point` type to make it a `record struct` as well:

```C#
public record struct Point(int X, int Y)
```

   A `record struct` is a `struct` type that includes the extra behavior added to all `record` types.

7. Try this version by typing `dotnet tuples.cs` in the terminal window.

Struct, class, and interface types

All concrete named types in C# are either `class` or `struct` types, including `record` types. A `class` is a *reference type*. A `struct` is a *value type*. Variables of a value type store the contents of the instance inline in memory. In other words, a `record struct Point` stores two integers: `X` and `Y`. Variables of a reference type store a reference, or pointer, to the storage for the instance. In other words, a `record class Point` stores a reference to a block of memory that holds the values for `X` and `Y`.

In practice, that difference means value types are copied when assigned, but a copy of a class instance is a copy of the *reference*. That copied reference refers to the same instance of a point, with the same storage for `X` and `Y`.

The `record` modifier instructs the compiler to write several members for you. You can learn more in the article on record types in the fundamentals section.

When you declare a `record` type, you declare that your type should use a default set of behaviors for equality comparisons, assignment, and copying instances of that type. Records are the best choice when storing related data is the primary responsibility of your type. As you add more behaviors, consider using `struct` or `class` types, without the `record` modifier.

Use `struct` types for value types when you need more sophisticated behavior, but the primary responsibility is storing values. Use `class` types to use object-oriented idioms like encapsulation, inheritance, and polymorphism.

You can also define `interface` types to declare behavioral contracts that different types must implement. Both `struct` and `class` types can implement interfaces.

You typically use all these types in larger programs and libraries. Once you install the SDK, you can explore those types using tutorials on classes in the fundamentals section.

You completed the "Create types in C#" tutorial.

Cleanup resources

GitHub automatically deletes your Codespace after 30 days of inactivity. If you plan to explore more tutorials in this series, you can leave your Codespace provisioned. If you're ready to visit the .NET site to download the .NET SDK, you can delete your Codespace. To delete your Codespace, open a browser window and navigate to your Codespaces. You should see a list of your codespaces in the window. Select the three dots (`...`) in the entry for the learn tutorial codespace and select **delete**.

Next steps

Continue to the next tutorial in this series:

Branches and loops

Or explore related topics in C# Fundamentals:

- The C# type system — Dive deeper into the types you used in this tutorial.

- Record types — Learn more about records and when to use them.
- Classes — Explore object-oriented programming in C#.
- What you can build with C# — See the kinds of apps you can create with what you're learning.

Last updated on 02/06/2026

Tutorial: C# `if` statements and loops - conditional logic

Summarize this article for me

This tutorial teaches you how to write C# code that examines variables and changes the execution path based on those variables. You write C# code and see the results of compiling and running it. The tutorial contains a series of lessons that explore branching and looping constructs in C#. These lessons teach you the fundamentals of the C# language.

> **Tip**
>
> **New to programming?** Work through every section in order. **Coming from another language?** The `if`/`else` and loop syntax looks familiar - focus on the nested loops and the challenge to practice C#-specific patterns.

In this tutorial, you:

- ✓ Launch a GitHub Codespace with a C# development environment.
- ✓ Explore `if` and `else` statements.
- ✓ Use loops to repeat operations.
- ✓ Work with nested loops.
- ✓ Combine branches and loops.

Prerequisites

You must have one of the following options:

- A GitHub account to use GitHub Codespaces. If you don't already have one, create a free account at GitHub.com.
- A computer with the following tools installed:
  - The .NET 10 SDK.
  - Visual Studio Code.
  - The C# DevKit.

Use if statements

To start a GitHub Codespace with the tutorial environment, open a browser window to the tutorial codespace repository. Select the green *Code* button, and the *Codespaces* tab. Then

select the `+` sign to create a new Codespace using this environment. If you completed other tutorials in this series, you can open that codespace instead of creating a new one.

1. When your codespace loads, create a new file in the *tutorials* folder named *branches-loops.cs*.

2. Open your new file.

3. Type or copy the following code into *branches-loops.cs*:

   C#

   ```
   int a = 5;
   int b = 6;
   if (a + b > 10)
       Console.WriteLine("The answer is greater than 10.");
   ```

4. Try this code by typing the following command in the integrated terminal:

   .NET CLI

   ```
   cd tutorials
   dotnet branches-loops.cs
   ```

   You should see the message "The answer is greater than 10." printed to your console.

5. Modify the declaration of `b` so that the sum is less than 10:

   C#

   ```
   int b = 3;
   ```

6. Type `dotnet branches-loops.cs` again in the terminal window.

   Because the answer is less than 10, nothing is printed. The **condition** you're testing is false. You don't have any code to execute because you only wrote one of the possible branches for an `if` statement: the true branch.

> **Tip**
>
> As you explore C# (or any programming language), you might make mistakes when you write code. The compiler finds and reports the errors. Look closely at the error output and the code that generated the error. You can also ask Copilot to find differences or spot any mistakes. The compiler error can usually help you find the problem.

This first sample shows the power of `if` and Boolean types. A *Boolean* is a variable that can have one of two values: `true` or `false`. C# defines a special type, `bool` for Boolean variables. The `if` statement checks the value of a `bool`. When the value is `true`, the statement following the `if` executes. Otherwise, it's skipped. This process of checking conditions and executing statements based on those conditions is powerful. Let's explore more.

Make if and else work together

To execute different code in both the true and false branches, create an `else` branch that runs when the condition is false. Try an `else` branch.

1. Add the last two lines in the following code snippet (you should already have the first four):

   C#

   ```
   int a = 5;
   int b = 3;
   if (a + b > 10)
       Console.WriteLine("The answer is greater than 10");
   else
       Console.WriteLine("The answer is not greater than 10");
   ```

   The statement following the `else` keyword executes only when the condition being tested is `false`. Combining `if` and `else` with boolean conditions provides all the power you need to handle both a `true` and a `false` condition.

   > **ⓘ Important**
   >
   > The indentation under the `if` and `else` statements is for human readers. The C# language doesn't treat indentation or white space as significant. The statement following the `if` or `else` keyword executes based on the condition. All the samples in this tutorial follow a common practice to indent lines based on the control flow of statements.

   Because indentation isn't significant, you need to use `{` and `}` to indicate when you want more than one statement to be part of the block that executes conditionally. C# programmers typically use those braces on all `if` and `else` clauses.

2. The following example is the same as what you created in the previous example, with the addition of `{` and `}`. Modify your code to match the following code:

C#

```
int a = 5;
int b = 3;
if (a + b > 10)
{
    Console.WriteLine("The answer is greater than 10");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
}
```

> **Tip**
>
> Through the rest of this tutorial, the code samples all include the braces, following accepted practices.

3. You can test more complicated conditions. Add the following code after the code you wrote so far:

C#

```
int a = 5;
int b = 3;
int c = 4;
if ((a + b + c > 10) && (a == b))
{
    Console.WriteLine("The answer is greater than 10");
    Console.WriteLine("And the first number is equal to the second");
}
else
{
    Console.WriteLine("The answer is not greater than 10");
    Console.WriteLine("Or the first number is not equal to the second");
}
```

   The `==` symbol tests for *equality*. Using `==` distinguishes the test for equality from assignment, which you saw in `a = 5`.

   The `&&` represents "and". It means both conditions must be true to execute the statement in the true branch. These examples also show that you can have multiple statements in each conditional branch, provided you enclose them in `{` and `}`.

4. You can also use `||` to represent "or". Add the following code after what you wrote so far:

C#

```
if ((a + b + c > 10) || (a == b))
```

5. Modify the values of `a`, `b`, and `c` and switch between `&&` and `||` to explore. You gain more understanding of how the `&&` and `||` operators work.

6. You finished the first step. Before you start the next section, move the current code into a separate method. That change makes it easier to start working with a new example. Put the existing code in a method called `ExploreIf()`. Call it from the top of your program. When you finish those changes, your code should look like the following code:

C#

```
ExploreIf();

void ExploreIf()
{
    int a = 5;
    int b = 3;
    if (a + b > 10)
    {
        Console.WriteLine("The answer is greater than 10");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
    }

    int c = 4;
    if ((a + b + c > 10) && (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("And the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("Or the first number is not greater than the
second");
    }

    if ((a + b + c > 10) || (a > b))
    {
        Console.WriteLine("The answer is greater than 10");
        Console.WriteLine("Or the first number is greater than the second");
    }
    else
    {
        Console.WriteLine("The answer is not greater than 10");
        Console.WriteLine("And the first number is not greater than the
second");
```

```
    }
}
```

7. Comment out the call to `ExploreIf()`. It makes the output less cluttered as you work in this section:

```C#
//ExploreIf();
```

The `//` starts a **comment** in C#. Comments are any text you want to keep in your source code but don't execute as code. The compiler doesn't generate any executable code from comments.

Use loops to repeat operations

Another important concept for creating larger programs is **loops**. Use loops to repeat statements that you want to execute more than once.

> **Tip**
>
> **Learn more:** Read about iteration statements and selection statements in the C# language reference.

1. Add this code after the call to `ExploreIf`:

```C#
int counter = 0;
while (counter < 10)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
}
```

The `while` statement checks a condition and executes the statement following the `while`. It repeats checking the condition and executing those statements until the condition is false.

There's one other new operator in this example. The `++` after the `counter` variable is the **increment** operator. It adds 1 to the value of `counter` and stores that value in the `counter` variable.

> **ⓘ Important**
>
> Make sure that the `while` loop condition changes to false as you execute the code. Otherwise, you create an **infinite loop** where your program never ends. This sample doesn't demonstrate that behavior, because you have to force your program to quit by using **CTRL-C** or other means.

The `while` loop tests the condition before executing the code following the `while`.

2. The `do` ... `while` loop executes the code first, and then checks the condition. The *do while* loop is shown in the following code:

```
C#

int counter = 0;
do
{
    Console.WriteLine($"Hello World! The counter is {counter}");
    counter++;
} while (counter < 10);
```

This `do` loop and the earlier `while` loop produce the same output.

Let's move on to one last loop statement.

Work with the for loop

Another common loop statement that you see in C# code is the `for` loop.

1. Try this code:

```
C#

for (int counter = 0; counter < 10; counter++)
{
    Console.WriteLine($"Hello World! The counter is {counter}");
}
```

The preceding `for` loop does the same work as the `while` loop and the `do` loop you already used. The `for` statement has three parts that control how it works:

- The first part is the **for initializer**: `int counter = 0;` declares that `counter` is the loop variable, and sets its initial value to `0`.

- The middle part is the **for condition**: `counter < 10` declares that this `for` loop continues to execute as long as the value of `counter` is less than 10.
- The final part is the **for iterator**: `counter++` specifies how to modify the loop variable after executing the block following the `for` statement. Here, it specifies that `counter` increments by 1 each time the block executes.

2. Experiment with these conditions yourself. Try each of the following changes:

   - Change the initializer to start at a different value.
   - Change the condition to stop at a different value.

When you're done, move on to the next section to write some code yourself and use what you learned.

There's one other looping statement that isn't covered in this tutorial: the `foreach` statement. The `foreach` statement repeats its statement for every item in a sequence of items. You most often use it with *collections*. It's covered in the next tutorial.

Created nested loops

You can nest a `while`, `do`, or `for` loop inside another loop. By combining each item in the outer loop with each item in the inner loop, you create a matrix. Let's build a set of alphanumeric pairs to represent rows and columns.

1. Add the following `for` loop that generates the rows:

```C#
for (int row = 1; row < 11; row++)
{
    Console.WriteLine($"The row is {row}");
}
```

2. Add another loop to generate the columns:

```C#
for (char column = 'a'; column < 'k'; column++)
{
    Console.WriteLine($"The column is {column}");
}
```

3. Finally, nest the columns loop inside the rows to form pairs:

C#

```
for (int row = 1; row < 11; row++)
{
    for (char column = 'a'; column < 'k'; column++)
    {
        Console.WriteLine($"The cell is ({row}, {column})");
    }
}
```

The outer loop increments once for each full run of the inner loop. Reverse the row and column nesting, and see the changes for yourself. When you're done, place the code from this section in a method called `ExploreLoops()`.

Combine branches and loops

Now that you used the `if` statement and the looping constructs in the C# language, see if you can write C# code to find the sum of all integers 1 through 20 that are divisible by 3. Here are a few hints:

- The `%` operator gives you the remainder of a division operation.
- The `if` statement gives you the condition to see if a number should be part of the sum.
- The `for` loop can help you repeat a series of steps for all the numbers 1 through 20.

Try it yourself. Then check how you did. As a hint, you should get 63 for an answer.

Did you come up with something like this?

▼ Details

C#

```
int sum = 0;
for (int number = 1; number < 21; number++)
{
    if (number % 3 == 0)
    {
        sum = sum + number;
    }
}
Console.WriteLine($"The sum is {sum}");
```

You completed the "branches and loops" tutorial.

Cleanup resources

GitHub automatically deletes your Codespace after 30 days of inactivity. If you plan to explore more tutorials in this series, you can leave your Codespace provisioned. If you're ready to visit the .NET site to download the .NET SDK, you can delete your Codespace. To delete your Codespace, open a browser window and navigate to your Codespaces. You should see a list of your codespaces in the window. Select the three dots (`...`) in the entry for the learn tutorial codespace and select **delete**.

Next steps

Continue to the next tutorial in this series:

Learn about collections

Or explore related topics in C# Fundamentals:

- Pattern matching — A powerful alternative to complex `if`/`else` chains.
- Methods and program structure — Learn how to organize the methods you created in this tutorial.
- What you can build with C# — See the kinds of apps you can create with what you're learning.
- Selection statements
- Iteration statements

Last updated on 02/09/2026

Tutorial: Learn to manage data collections using List<T> in C#

Summarize this article for me

This introductory tutorial provides an introduction to the C# language and the basics of the System.Collections.Generic.List<T> class.

This tutorial teaches you C#. You write C# code and see the results of compiling and running that code. It contains a series of lessons that create, modify, and explore collections. You work primarily with the List<T> class.

> **Tip**
>
> **New to programming?** Work through each section in order. **Coming from another language?** If you already know dynamic arrays or lists, skim the basics and focus on search and sort, lists of other types, and the challenge.

In this tutorial, you:

- ✓ Launch a GitHub Codespace with a C# development environment.
- ✓ Create different types of lists.
- ✓ Modify list contents.
- ✓ Search and sort lists.

Prerequisites

You must have one of the following options:

- A GitHub account to use GitHub Codespaces. If you don't already have one, you can create a free account at GitHub.com.
- A computer with the following tools installed:
  - The .NET 10 SDK.
  - Visual Studio Code.
  - The C# DevKit.

A basic list example

To start a GitHub Codespace with the tutorial environment, open a browser window to the tutorial codespace repository. Select the green *Code* button, and select the *Codespaces* tab.

Then select the `+` sign to create a new Codespace using this environment. If you completed other tutorials in this series, you can open that codespace instead of creating a new one.

1. When your codespace loads, create a new file in the *tutorials* folder named *lists.cs*.
2. Open your new file.
3. Type or copy the following code into *lists.cs*:

   C#

   ```
   List<string> names = ["<name>", "Ana", "Felipe"];
   foreach (var name in names)
   {
       Console.WriteLine($"Hello {name.ToUpper()}!");
   }
   ```

4. Run the code by typing the following command in the terminal window:

   .NET CLI

   ```
   cd tutorials
   dotnet lists.cs
   ```

You created a list of strings, added three names to that list, and printed the names in all caps. You're using concepts that you learned in earlier tutorials to loop through the list.

The code to display names makes use of the string interpolation feature. When you precede a `string` with the `$` character, you can embed C# code in the string declaration. The actual string replaces that C# code with the value it generates. In this example, it replaces the `{name.ToUpper()}` with each name, converted to capital letters, because you called the String.ToUpper method.

Let's keep exploring.

Modify list contents

The collection you created uses the List<T> type. This type stores sequences of elements. You specify the type of the elements between the angle brackets.

> **Tip**
>
> **Learn more:** Explore collections and generics in the C# documentation.

One important aspect of this List<T> type is that it can grow or shrink, so you can add or remove elements. You can see the results by modifying the contents after you displayed its contents.

1. Add the following code after the code you already wrote (the loop that prints the contents):

   C#

   ```
   Console.WriteLine();
   names.Add("Maria");
   names.Add("Bill");
   names.Remove("Ana");
   foreach (var name in names)
   {
       Console.WriteLine($"Hello {name.ToUpper()}!");
   }
   ```

   You added two more names to the end of the list. You also removed one as well. The output from this block of code shows the initial contents, then prints a blank line and the new contents.

2. The List<T> type also enables you to reference individual items by **index**. You access items by using the `[` and `]` tokens. Add the following code after what you already wrote and try it:

   C#

   ```
   Console.WriteLine($"My name is {names[0]}.");
   Console.WriteLine($"I've added {names[2]} and {names[3]} to the list.");
   ```

   You can't access past the end of the list. You can check how long the list is by using the Count property.

3. Add the following code:

   C#

   ```
   Console.WriteLine($"The list has {names.Count} people in it");
   ```

   Type `dotnet lists.cs` again in the terminal window to see the results. In C#, indices start at 0, so the largest valid index is one less than the number of items in the list.

For more information about indices, see the Explore indexes and ranges article.

Search and sort lists

The samples use relatively small lists, but your applications might often create lists with many more elements, sometimes numbering in the thousands. To find elements in these larger collections, you need to search the list for different items. The IndexOf method searches for an item and returns the index of the item. If the item isn't in the list, `IndexOf` returns `-1`.

1. Try it to see how it works. Add the following code after what you wrote so far:

   C#

   ```
   var index = names.IndexOf("Felipe");
   if (index == -1)
   {
       Console.WriteLine($"When an item is not found, IndexOf returns {index}");
   }
   else
   {
       Console.WriteLine($"The name {names[index]} is at index {index}");
   }

   index = names.IndexOf("Not Found");
   if (index == -1)
   {
       Console.WriteLine($"When an item is not found, IndexOf returns {index}");
   }
   else
   {
       Console.WriteLine($"The name {names[index]} is at index {index}");
   }
   ```

   You might not know if an item is in the list, so always check the index returned by IndexOf. If it's `-1`, the item wasn't found.

2. You can also sort the items in your list. The Sort method sorts all the items in the list in their normal order (alphabetically for strings). Add this code and run it again:

   C#

   ```
   names.Sort();
   foreach (var name in names)
   {
       Console.WriteLine($"Hello {name.ToUpper()}!");
   }
   ```

Lists of other types

So far, you've been using the `string` type in lists. Let's make a List<T> using a different type. Let's build a set of numbers.

1. Add the following code at the end of your source file:

   C#

   ```
   List<int> fibonacciNumbers = [1, 1];
   ```

   That code creates a list of integers and sets the first two integers to the value 1. The *Fibonacci Sequence*, a sequence of numbers, starts with two 1s. Each next Fibonacci number is found by taking the sum of the previous two numbers.

2. Add this code:

   C#

   ```
   var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
   var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];

   fibonacciNumbers.Add(previous + previous2);

   foreach (var item in fibonacciNumbers)
   {
       Console.WriteLine(item);
   }
   ```

3. Type `dotnet lists.cs` in the terminal window to see the results.

Challenge

See if you can put together some of the concepts from this and earlier lessons. Expand on what you built so far with Fibonacci numbers. Try to write the code to generate the first 20 numbers in the sequence. (As a hint, the 20th Fibonacci number is 6765.)

Did you come up with something like this code?

▼ Details

C#

```
List<int> fibonacciNumbers = [1, 1];

while (fibonacciNumbers.Count < 20)
{
    var previous = fibonacciNumbers[fibonacciNumbers.Count - 1];
    var previous2 = fibonacciNumbers[fibonacciNumbers.Count - 2];
```

```
    fibonacciNumbers.Add(previous + previous2);
}
foreach (var item in fibonacciNumbers)
{
    Console.WriteLine(item);
}
```

With each iteration of the loop, you take the last two integers in the list, sum them, and add that value to the list. The loop repeats until you add 20 items to the list.

You completed the list tutorial.

Cleanup resources

GitHub automatically deletes your Codespace after 30 days of inactivity. If you plan to explore more tutorials in this series, you can leave your Codespace provisioned. If you're ready to visit the .NET site to download the .NET SDK, you can delete your Codespace. To delete your Codespace, open a browser window and navigate to your Codespaces. You should see a list of your codespaces in the window. Select the three dots (`...`) in the entry for the learn tutorial codespace and select **delete**.

Next steps

Continue to the next tutorial in this series:

Pattern matching

Or explore related topics in C# Fundamentals:

- Generics in C# — Understand the `<T>` syntax you used throughout this tutorial.
- LINQ overview — Query and transform collections by using Language Integrated Query.
- What you can build with C# — See the kinds of apps you can create with what you're learning.
- Selecting a collection type
- Commonly used collection types
- When to use generic collections

Last updated on 02/23/2026

Tutorial: Use C# to match data against patterns

Summarize this article for me

This tutorial teaches you how to use pattern matching to inspect data in C#. You write small amounts of code, then you compile and run that code. The tutorial contains a series of lessons that explore different kinds of patterns supported by C#. These lessons teach you the fundamentals of the C# language.

> **Tip**
>
> **New to programming?** Work through each section in order. **Coming from another language?** If you already know `switch` statements, focus on exhaustive matches and type patterns – these are distinctive C# features.

In this tutorial, you:

- ✓ Launch a GitHub Codespace with a C# development environment.
- ✓ Test data for discrete values.
- ✓ Match enum data against value.
- ✓ Create exhaustive matches using `switch` expressions.
- ✓ Match types using type patterns.

Prerequisites

You must have one of the following options:

- A GitHub account to use GitHub Codespaces. If you don't already have one, you can create a free account at GitHub.com.
- A computer with the following tools installed:
  - The .NET 10 SDK.
  - Visual Studio Code.
  - The C# DevKit.

Match a value

The preceding tutorials demonstrated built-in types and types you define as tuples or records. You can check instances of these types against a *pattern*. Whether an instance matches a pattern determines the actions your program takes. In the examples that follow, you see `?`

after type names. This symbol allows the value of this type to be null (for example, `bool?` can be `true`, `false`, or `null`). For more information, see Nullable value types. Let's start to explore how you can use patterns.

To start a GitHub Codespace with the tutorial environment, open a browser window to the tutorial codespace repository. Select the green *Code* button, and the *Codespaces* tab. Then select the `+` sign to create a new Codespace using this environment. If you completed other tutorials in this series, you can open that codespace.

1. When your codespace loads, create a new file in the *tutorials* folder named *patterns.cs*.

2. Open your new file.

3. All the examples in this tutorial use text input that represents a series of bank transactions as comma separated values (CSV) input. In each of the samples, you can match the record against a pattern by using either an `is` or `switch` expression. This first example splits each line on the `,` character and then *matches* the first string field against the value "DEPOSIT" or "WITHDRAWAL" by using an `is` expression. When it matches, the transaction amount is added or deducted from the current account balance. To see it work, add the following code to *patterns.cs*:

   C#

   ```
   string bankRecords = """
       DEPOSIT,   10000, Initial balance
       DEPOSIT,     500, regular deposit
       WITHDRAWAL, 1000, rent
       DEPOSIT,    2000, freelance payment
       WITHDRAWAL,  300, groceries
       DEPOSIT,     700, gift from friend
       WITHDRAWAL,  150, utility bill
       DEPOSIT,    1200, tax refund
       WITHDRAWAL,  500, car maintenance
       DEPOSIT,     400, cashback reward
       WITHDRAWAL,  250, dining out
       DEPOSIT,    3000, bonus payment
       WITHDRAWAL,  800, loan repayment
       DEPOSIT,     600, stock dividends
       WITHDRAWAL,  100, subscription fee
       DEPOSIT,    1500, side hustle income
       WITHDRAWAL,  200, fuel expenses
       DEPOSIT,     900, refund from store
       WITHDRAWAL,  350, shopping
       DEPOSIT,    2500, project milestone payment
       WITHDRAWAL,  400, entertainment
       """;

   double currentBalance = 0.0;
   var reader = new StringReader(bankRecords);
   ```

```
string? line;
while ((line = reader.ReadLine()) is not null)
{
    if (string.IsNullOrWhiteSpace(line)) continue;
    // Split the line based on comma delimiter and trim each part
    string[] parts = line.Split(',');

    string? transactionType = parts[0]?.Trim();
    if (double.TryParse(parts[1].Trim(), out double amount))
    {
        // Update the balance based on transaction type
        if (transactionType?.ToUpper() is "DEPOSIT")
            currentBalance += amount;
        else if (transactionType?.ToUpper() is "WITHDRAWAL")
            currentBalance -= amount;

        Console.WriteLine($"{line.Trim()} => Parsed Amount: {amount}, New
Balance: {currentBalance}");
    }
}
```

4. Then, type the following text in the terminal window:

.NET CLI

```
cd tutorials
dotnet patterns.cs
```

5. Examine the output. You can see that each line is processed by comparing the value of the text in the first field.

You could similarly construct the preceding sample by using the `==` operator to test that two `string` values are equal. Comparing a variable to a constant is a basic building block for pattern matching. Let's explore more of the building blocks that are part of pattern matching.

> **Tip**
>
> **Learn more:** Read about pattern matching in the C# Fundamentals section for a comprehensive overview of all pattern types.

Enum matches

Another common use for pattern matching is matching on the values of an `enum` type. The following sample processes the input records to create a *tuple* where the first value is an `enum` value that notes a deposit or a withdrawal. The second value is the value of the transaction.

1. Add the following code to the end of your source file. It defines the `TransactionType` enumeration:

```C#
public enum TransactionType
{
    Deposit,
    Withdrawal,
    Invalid
}
```

2. Add a function to parse a bank transaction into a tuple that holds the transaction type and the value of the transaction. Add the following code before your declaration of the `TransactionType` enum:

```C#
static IEnumerable<(TransactionType type, double amount)>
TransactionRecords(string inputText)
{
    var reader = new StringReader(inputText);
    string? line;
    while ((line = reader.ReadLine()) is not null)
    {
        string[] parts = line.Split(',');

        string? transactionType = parts[0]?.Trim();
        if (double.TryParse(parts[1].Trim(), out double amount))
        {
            // Update the balance based on transaction type
            if (transactionType?.ToUpper() is "DEPOSIT")
                yield return (TransactionType.Deposit, amount);
            else if (transactionType?.ToUpper() is "WITHDRAWAL")
                yield return (TransactionType.Withdrawal, amount);
        }
        else {
        yield return (TransactionType.Invalid, 0.0);
        }
    }
}
```

3. Add a new loop to process the transaction data by using the `TransactionType` enumeration you declared:

```C#
currentBalance = 0.0;
```

```
foreach (var transaction in TransactionRecords(bankRecords))
{
    if (transaction.type == TransactionType.Deposit)
        currentBalance += transaction.amount;
    else if (transaction.type == TransactionType.Withdrawal)
        currentBalance -= transaction.amount;
    Console.WriteLine($"{transaction.type} => Parsed Amount:
{transaction.amount}, New Balance: {currentBalance}");
}
```

The preceding example also uses an `if` statement to check the value of an `enum` expression. Another form of pattern matching uses a `switch` expression. Let's explore that syntax and how you can use it.

Exhaustive matches with `switch`

A series of `if` statements can test a series of conditions. But, the compiler can't tell if a series of `if` statements are *exhaustive* or if later `if` conditions are *subsumed* by earlier conditions. *Exhaustive* means that one of the `if` or `else` clauses in the series of tests handles all possible inputs. If a series of `if` statements are exhaustive, every possible input satisfies at least one `if` or `else` clause. *Subsumption* means that a later `if` or `else` clause can't be reached because earlier `if` or `else` clauses match all possible inputs. For example, in the following example code, one clause never matches:

C#

```
int n = GetNumber();

if (n < 20)
    Console.WriteLine("n is less than 20");
else if (n < 10)
    Console.WriteLine("n is less than 10"); // unreachable
else
    Console.WriteLine("n is greater than 20");
```

The `else if` clause never matches because every number less than 10 is also less than 20. The `switch` expression ensures both of those characteristics are met, which results in fewer bugs in your apps. Let's try it and experiment.

1. Copy the following code. Replace the two `if` statements in your `foreach` loop with the `switch` expression you copied.

   C#

```
currentBalance += transaction switch
{
    (TransactionType.Deposit, var amount) => amount,
    (TransactionType.Withdrawal, var amount) => -amount,
    _ => 0.0,
};
```

2. Type `dotnet patterns.cs` in the terminal window to run the new sample.

   When you run the code, you see that it works the same.

3. To demonstrate *subsumption*, reorder the switch arms as shown in the following snippet:

   C#

   ```
   currentBalance += transaction switch
   {
       (TransactionType.Deposit, var amount) => amount,
       _ => 0.0,
       (TransactionType.Withdrawal, var amount) => -amount,
   };
   ```

   After you reorder the switch arms, type `dotnet patterns.cs` in the terminal window. The compiler reports an error because the arm with `_` matches every value. As a result, that final arm with `TransactionType.Withdrawal` never runs. The compiler tells you that something's wrong in your code.

   The compiler reports a warning if the expression tested in a `switch` expression could contain values that don't match any switch arm. If some values can't match any condition, the `switch` expression isn't *exhaustive*. The compiler also reports a warning if some values of the input don't match any of the switch arms.

4. Remove the line with `_ => 0.0,`, so that any invalid values don't match.

5. Type `dotnet patterns.cs` to see the results.

   The compiler reports a warning. The test data is valid, so the program works. However, any invalid data would cause a failure at runtime.

Type patterns

To finish this tutorial, explore one more building block for pattern matching: the *type pattern*. A *type pattern* tests an expression at run time to see if it's the specified type. You can use a type test with either an `is` expression or a `switch` expression. Modify the current sample in two

ways. First, instead of a tuple, build `Deposit` and `Withdrawal` record types that represent the transactions.

1. Add the following declarations at the end of your code file:

```C#
public record Deposit(double Amount, string description);
public record Withdrawal(double Amount, string description);
```

2. Add this method just before the declaration of the `TransactionType` enumeration. It parses the text and returns a series of records:

```C#
static IEnumerable<object?> TransactionRecordType(string inputText)
{
    var reader = new StringReader(inputText);
    string? line;
    while ((line = reader.ReadLine()) is not null)
    {
        string[] parts = line.Split(',');

        string? transactionType = parts[0]?.Trim();
        if (double.TryParse(parts[1].Trim(), out double amount))
        {
            // Update the balance based on transaction type
            if (transactionType?.ToUpper() is "DEPOSIT")
                yield return new Deposit(amount, parts[2]);
            else if (transactionType?.ToUpper() is "WITHDRAWAL")
                yield return new Withdrawal(amount, parts[2]);
        }
    }
}
```

3. Add the following code after the last `foreach` loop:

```C#
currentBalance = 0.0;

foreach (var transaction in TransactionRecordType(bankRecords))
{
    currentBalance += transaction switch
    {
        Deposit d => d.Amount,
        Withdrawal w => -w.Amount,
        _ => 0.0,
    };
```

```
    Console.WriteLine($" {transaction} => New Balance: {currentBalance}");
}
```

4. Type `dotnet patterns.cs` in the terminal window to see the results. This final version tests the input against a *type*.

Pattern matching provides a vocabulary to compare an expression against characteristics. Patterns can include the expression's type, values of types, property values, and combinations of them. Comparing expressions against a pattern can be clearer than multiple `if` comparisons. You explored some of the patterns you can use to match expressions. There are many more ways to use pattern matching in your applications. As you explore, you can learn more about pattern matching in C# in the following articles:

- Pattern matching in C#
- Explore pattern matching tutorial
- Pattern matching scenario
- The C# type system — Understand the types you matched against in this tutorial.

Cleanup resources

GitHub automatically deletes your Codespace after 30 days of inactivity. You completed all the tutorials in this series. To delete your Codespace now, open a browser window and go to your Codespaces. You should see a list of your codespaces in the window. Select the three dots (`...`) in the entry for the learn tutorial codespace and select **delete**.

Next steps

You completed all the introductory tutorials! Here's where to go next:

- File-based apps — Learn about the `dotnet run` command you used throughout these tutorials.
- C# Fundamentals — Dive deeper into program structure, types, and object-oriented programming.
- What you can build with C# — See the kinds of apps you can create with what you learned.
- Download and install the .NET 10 SDK.
- Download and install Visual Studio Code.
- Download and install the C# DevKit.

Last updated on 02/23/2026

Annotated C# strategy

Summarize this article for me

> **Tip**
>
> This article is most useful for professionals and students who want to understand C#'s design philosophy and how the language evolves. If you're new to C#, start with A tour of C# or the beginner tutorials instead.

We will keep evolving C# to meet the changing needs of developers and remain a state-of-the-art programming language. We will innovate eagerly and broadly in collaboration with the teams responsible for .NET libraries, developer tools, and workload support, while being careful to stay within the spirit of the language. Recognizing the diversity of domains where C# is being used, we will prefer language and performance improvements that benefit all or most developers and maintain a high commitment to backwards compatibility. We will continue to empower the broader .NET ecosystem and grow its role in C#'s future, while maintaining stewardship of design decisions.

How strategy guides C#

The C# strategy guides our decisions about C# evolution, and these annotations provide insight into how we think about key statements.

> "we will innovate eagerly and broadly"

The C# community continues to grow, and the C# language continues to evolve to meet the community's needs and expectations. We draw inspiration from many sources to select features that benefit a large segment of C# developers, and that provide consistent improvements in productivity, readability, and performance.

> "being careful to stay within the spirit of the language"

We evaluate new ideas in the spirit and history of the C# language. We prioritize innovations that make sense to most existing C# developers.

> "improvements that benefit all or most developers"

Developers use C# in all .NET workloads. Developers build web front and back ends, cloud native apps, and desktop apps with C#. C# enables cross platform applications. We focus on

new features that have the most impact either directly, or by empowering improvements to common libraries. Language feature development includes integration into our developer tools and learning resources.

> "high commitment to backwards compatibility"

We respect that there's a massive amount of C# code in use today. Any potential breaking change is carefully considered against the scale and impact of disruption to the C# community.

> "maintaining stewardship"

C# language design takes place in the open with community participation. Anyone can propose new C# features in our GitHub repos. The Language Design Team makes the final decisions after weighing community input.

Last updated on 02/23/2026

Roadmap for Java developers learning C#

Summarize this article for me

If you're starting a new job or joining a team that uses C#, this article helps you get productive quickly. It highlights what's familiar from Java and what's new in C#.

C# and Java have many similarities. As you learn C#, you can apply much of the knowledge you already have from programming in Java:

1. ***Similar syntax***: Both Java and C# are in the C family of languages. That similarity means you can already read and understand C#. There are some differences, but most of the syntax is the same as Java, and C. The curly braces and semicolons are familiar. The control statements like `if`, `else`, `switch` are the same. The looping statements of `for`, `while`, and `do`...`while` are same. The same keywords for `class` and `interface` are in both languages. The access modifiers from `public` to `private` are the same. Even many of the built-in types use the same keywords: `int`, `string`, and `double`.
2. ***Object-oriented paradigm***: Both Java and C# are object-oriented languages. The concepts of polymorphism, abstraction, and encapsulation apply in both languages. Both added new constructs, but the core features are still relevant.
3. ***Strongly typed***: Both Java and C# are strongly typed languages. You declare the data type of variables, either explicitly or implicitly. The compiler enforces type safety. The compiler catches type-related errors in your code, before you run the code.
4. ***Cross-platform***: Both Java and C# are cross-platform. You can run your development tools on your favorite platform. Your application can run on multiple platforms. Your development platform isn't required to match your target platform.
5. ***Exception handling***: Both Java and C# throw exceptions to indicate errors. Both use `try` - `catch` - `finally` blocks to handle exceptions. The Exception classes have similar names and inheritance hierarchies. One difference is that C# doesn't have the concept of *checked exceptions*. Any method might (in theory) throw any exception.
6. ***Standard libraries***: The .NET runtime and the Java Standard Library (JSL) have support for common tasks. Both have extensive ecosystems for other open source packages. In C#, the package manager is NuGet. It's analogous to Maven.
7. ***Garbage Collection***: Both languages employ automatic memory management through garbage collection. The runtime reclaims the memory from objects that aren't referenced. One difference is that C# enables you to create value types, as `struct` types.

Syntax at a glance

The following examples show a few common patterns side by side. These comparisons aren't exhaustive, but they give you a quick feel for the syntax differences.

Variable declaration and type inference:

Java

```java
// Java
var name = "Hello";
final int count = 5;
```

C#

```csharp
// C#
var name = "Hello";
const int count = 5;
```

String interpolation:

Java

```java
// Java
var message = "Hello, " + name + "! Count: " + count;
```

C#

```csharp
// C#
var message = $"Hello, {name}! Count: {count}";
```

Learn more: String interpolation

Lambda expressions:

Java

```java
// Java
list.stream().filter(x -> x > 5).collect(Collectors.toList());
```

C#

```csharp
// C#
var result = list.Where(x => x > 5).ToList();
```

Learn more: LINQ overview

Null handling:

Java

```
// Java
String value = optional.orElse("default");
```

C#

```
// C#
string value = input ?? "default";
```

Learn more: Nullable reference types

What's familiar

You can work productively in C# almost immediately because of the similarities. As you progress, learn features and idioms in C# that aren't available in Java:

1. ***Pattern matching***: Pattern matching enables concise conditional statements and expressions based on the shape of complex data structures. The is statement checks if a variable "is" some pattern. The pattern-based switch expression provides a rich syntax to inspect a variable and make decisions based on its characteristics.
2. ***String interpolation*** and ***raw string literals***: String interpolation enables you to insert evaluated expressions in a string, rather than using positional identifiers. Raw string literals provide a way to minimize escape sequences in text.
3. ***Nullable and non-nullable types***: C# supports *nullable value types*, and *nullable reference types* by appending the `?` suffix to a type. For nullable types, the compiler warns you if you don't check for `null` before dereferencing the expression. For non-nullable types, the compiler warns you if you might be assigning a `null` value to that variable. Non-nullable reference types minimize programming errors that throw a System.NullReferenceException.
4. ***Extensions***: In C#, you can create members that *extend* a class or interface. Extensions provide new behavior for a type from a library, or all types that implement a given interface.
5. ***LINQ***: Language integrated query (LINQ) provides a common syntax to query and transform data, regardless of its storage.
6. ***Local functions***: In C#, you can nest functions inside methods, or other local functions. Local functions provide yet another layer of encapsulation.

> **Tip**
>
> To learn more about C#'s type system—including `struct` vs. `class`, records, and interfaces—visit the **Type system** overview in the Fundamentals section.

There are other features in C# that aren't in Java. Features like async and await model asynchronous operations in sequential syntax. The using statement automatically frees nonmemory resources.

There are also some similar features between C# and Java that have subtle but important differences:

1. ***Properties*** and ***Indexers***: Both properties and indexers (treating a class like an array or dictionary) have language support. In Java, they're naming conventions for methods starting with `get` and `set`.
2. ***Records***: In C#, records can be either `class` (reference) or `struct` (value) types. C# records can be immutable, but aren't required to be immutable.
3. ***Tuples*** have different syntax in C# and Java.
4. ***Attributes*** are similar to Java annotations.

Finally, there are Java language features that aren't available in C#:

1. ***Checked exceptions***: In C#, any method could theoretically throw any exception.
2. ***Checked array covariance***: In C#, arrays aren't safely covariant. You should use the generic collection classes and interfaces if you need covariant structures.

Overall, learning C# for a developer experienced in Java should be smooth. C# has enough familiar idioms for you to be productive as you learn the new idioms.

Next steps

- A tour of C#: Get a high-level overview of all C# features.
- Beginner tutorials: Learn C# step by step with interactive lessons.
- What you can build with C#: Explore the application types you can create with C#.
- C# fundamentals: Dive deeper into the type system, object-oriented programming, and more.

Last updated on 02/23/2026

Roadmap for JavaScript and TypeScript developers learning C#

If you're joining a team that uses C#, or exploring it for server-side or full-stack development, this article helps you get productive quickly. It highlights what's familiar from JavaScript and TypeScript and what's new in C#.

C#, TypeScript, and JavaScript are all members of the C family of languages. The similarities between the languages help you quickly become productive in C#.

1. ***Similar syntax***: JavaScript, TypeScript, and C# are in the C family of languages. That similarity means you can already read and understand C#. There are some differences, but most of the syntax is the same as JavaScript and C. The curly braces and semicolons are familiar. The control statements like `if`, `else`, and `switch` are the same. The looping statements of `for`, `while`, and `do...while` are the same. The same keywords for `class` and `interface` are in both C# and TypeScript. The access modifiers in TypeScript and C#, from `public` to `private`, are the same.
2. ***The `=>` token***: All languages support lightweight function definitions. In C#, they're referred to as *lambda expressions*. In JavaScript, they're typically called *arrow functions*.
3. ***Function hierarchies***: All three languages support local functions, which are functions defined in other functions.
4. ***Async / Await***: All three languages share the same `async` and `await` keywords for asynchronous programming.
5. ***Garbage collection***: All three languages rely on a garbage collector for automatic memory management.
6. ***Event model***: C#'s event syntax is similar to JavaScript's model for document object model (DOM) events.
7. ***Package manager***: NuGet is the most common package manager for C#, similar to npm for JavaScript applications. C# libraries are delivered in assemblies.

Syntax at a glance

The following examples show a few common patterns side by side. These comparisons aren't exhaustive, but they give you a quick feel for the syntax differences.

Type annotations:

TypeScript

```
// TypeScript
let name: string = "Hello";
```

```
let count: number = 5;
```

C#

```
// C#
string name = "Hello";
int count = 5;
```

Async / await:

TypeScript

```
// TypeScript
async function fetchData(): Promise<string> {
    const response = await fetch(url);
    return await response.text();
}
```

C#

```
// C#
async Task<string> FetchDataAsync() {
    var response = await client.GetAsync(url);
    return await response.Content.ReadAsStringAsync();
}
```

To learn more, see Asynchronous programming.

Classes:

TypeScript

```
// TypeScript
class Point {
    constructor(public x: number, public y: number) {}
}
```

C#

```
// C#
record Point(int X, int Y);
```

To learn more, see Records.

Pattern matching:

TypeScript

```
// TypeScript - manual type checking
if (typeof value === "string") { /* ... */ }
```

C#

```
// C# - pattern matching
if (value is string s) { /* use s */ }
```

To learn more, see Pattern matching.

What's new for you in C#

As you learn C#, you encounter concepts that aren't part of JavaScript. Some of these concepts might be familiar to you if you use TypeScript:

1. ***C# Type System***: C# is a strongly typed language. Every variable has a type, and that type can't change. You define `class` or `struct` types. You can define interface definitions that define behavior implemented by other types. TypeScript includes many of these concepts, but because TypeScript is built on JavaScript, the type system isn't as strict.
2. ***Pattern matching***: Pattern matching enables concise conditional statements and expressions based on the shape of complex data structures. The is expression checks if a variable "is" some pattern. The pattern-based switch expression provides a rich syntax to inspect a variable and make decisions based on its characteristics.
3. ***String interpolation*** and ***raw string literals***: String interpolation enables you to insert evaluated expressions in a string, rather than using positional identifiers. Raw string literals provide a way to minimize escape sequences in text.
4. ***Nullable and non-nullable types***: C# supports *nullable value types* and *nullable reference types* by appending the `?` suffix to a type. For nullable types, the compiler warns you if you don't check for `null` before dereferencing the expression. For non-nullable types, the compiler warns you if you might be assigning a `null` value to that variable. These features can minimize your application throwing a System.NullReferenceException. The syntax might be familiar from TypeScript's use of `?` for optional properties.
5. ***LINQ***: Language integrated query (LINQ) provides a common syntax to query and transform data, regardless of its storage.

> **Tip**
>
> To learn more about C#'s type system—including `class` vs. `struct`, generics, and interfaces—visit the **Type system** overview in the Fundamentals section.

As you learn more, other differences become apparent, but many of those differences are smaller in scope.

Some familiar features and idioms from JavaScript and TypeScript aren't available in C#:

1. ***dynamic types***: C# uses static typing. A variable declaration includes the type, and that type can't change. There's a dynamic type in C# that provides runtime binding.
2. ***Prototypal inheritance***: C# inheritance is part of the type declaration. A C# `class` declaration states any base class. In JavaScript, you set the `__proto__` property to set the base type on any instance.
3. ***Interpreted language***: C# code must be compiled before you run it. JavaScript code can be run directly in the browser.

In addition, a few more TypeScript features aren't available in C#:

1. ***Union types***: Beginning with C# 15, C# supports union types. A union defines a closed set of named cases that a value can represent, and the compiler ensures exhaustive pattern matching over those cases.
2. ***Decorators***: C# doesn't have decorators. Some common decorators, such as `@sealed` are reserved keywords in C#. Other common decorators might have corresponding Attributes. For other decorators, you can create your own attributes.
3. ***More forgiving syntax***: The C# compiler parses code more strictly than JavaScript requires.

If you're building a web application, consider using Blazor to build your application. Blazor is a full-stack web framework built for C#. Blazor components can run on the server, as .NET assemblies, or on the client by using WebAssembly. Blazor supports interop with your favorite JavaScript or TypeScript libraries.

Next steps

- A tour of C#: Get a high-level overview of all C# features.
- Beginner tutorials: Learn C# step by step with interactive lessons.
- What you can build with C#: Explore the application types you can create with C#.
- C# fundamentals: Dive deeper into the type system, object-oriented programming, and more.

Last updated on 03/24/2026

Roadmap for Python developers learning C#

Summarize this article for me

If you're moving from Python to C# for a new role or project, this article helps you get productive quickly. It highlights what's familiar from Python and what's different in C#.

C# and Python share similar concepts. These familiar constructs help you learn C# when you already know Python.

1. ***Object oriented***: Both Python and C# are object-oriented languages. All the concepts around classes in Python apply in C#, even if the syntax is different.
2. ***Cross-platform***: Both Python and C# are cross-platform languages. Apps written in either language can run on many platforms.
3. ***Garbage collection***: Both languages employ automatic memory management through garbage collection. The runtime reclaims the memory from objects that aren't referenced.
4. ***Strongly typed***: Both Python and C# are strongly typed languages. Type coercion doesn't occur implicitly. There are differences described later, as C# is statically typed whereas Python is dynamically typed.
5. ***Async / Await***: Python's `async` and `await` feature was directly inspired by C#'s `async` and `await` support.
6. ***Pattern matching***: Python's `match` expression and pattern matching is similar to C#'s pattern matching `switch` expression. You use them to inspect a complex data expression to determine if it matches a pattern.
7. ***Statement keywords***: Python and C# share many keywords, such as `if`, `else`, `while`, `for`, and many others. While not all syntax is the same, there's enough similarity that you can read C# if you know Python.

Syntax at a glance

The following examples show a few common patterns side by side. These comparisons aren't exhaustive, but they give you a quick feel for the syntax differences.

Type annotations:

Python

```
# Python
name: str = "Hello"
count: int = 5
```

C#

```
// C#
string name = "Hello";
int count = 5;
```

List filtering (comprehension vs. LINQ):

Python

```
# Python
result = [x for x in items if x > 5]
```

C#

```
// C#
var result = items.Where(x => x > 5).ToList();
```

Learn more: LINQ overview

Block scope (indentation vs. braces):

Python

```
# Python
if count > 0:
    print("positive")
```

C#

```
// C#
if (count > 0)
{
    Console.WriteLine("positive");
}
```

Class definition:

Python

```
# Python
class Point:
    def __init__(self, x: int, y: int):
        self.x = x
        self.y = y
```

C#

```
// C#
record Point(int X, int Y);
```

Learn more: Records

Key differences

As you learn C#, you discover these important concepts where C# is different from Python:

1. ***Indentation vs. tokens***: In Python, newlines and indentation are first-class syntactic elements. In C#, whitespace isn't significant. Tokens, like `;` separate statements, and other tokens `{` and `}` control block scope for `if` and other block statements. However, for readability, most coding styles (including the style used in these docs) use indentation to reinforce the block scopes declared by `{` and `}`.
2. ***Static typing***: In C#, a variable declaration includes its type. Reassigning a variable to an object of a different type generates a compiler error. In Python, the type can change when reassigned.
3. ***Nullable types***: C# variables can be *nullable* or *non-nullable*. A non-nullable type is one that can't be null (or nothing). It always refers to a valid object. By contrast, a nullable type might either refer to a valid object, or null.
4. ***LINQ***: The query expression keywords that make up Language Integrated Query (LINQ) aren't keywords in Python. However, Python libraries like `itertools`, `more-itertools`, and `py-linq` provide similar functionality.
5. ***Generics***: C# generics use C# static typing to make assertions about the arguments supplied for type parameters. A generic algorithm might need to specify constraints that an argument type must satisfy.

> **Tip**
>
> To learn more about C#'s type system—including `class` vs. `struct`, generics, and interfaces—visit the Type system overview in the Fundamentals section.

Finally, some features of Python aren't available in C#:

1. ***Structural (duck) typing***: In C#, types have names and declarations. Except for tuples, types with the same structure aren't interchangeable.
2. ***REPL***: C# doesn't have a Read-Eval-Print Loop (REPL) to quickly prototype solutions.
3. ***Significant whitespace***: You need to correctly use braces `{` and `}` to note block scope.

Learning C# if you know Python is a smooth journey. The languages have similar concepts and similar idioms to use.

Next steps

- A tour of C#: Get a high-level overview of all C# features.
- Beginner tutorials: Learn C# step by step with interactive lessons.
- What you can build with C#: Explore the application types you can create with C#.
- C# fundamentals: Dive deeper into the type system, object-oriented programming, and more.

Last updated on 02/23/2026

General structure of a C# program

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. They walk you through writing your first C# programs before you learn about program structure.
>
> **Experienced in another language?** You might want to skim the Get started section for C#-specific syntax, then come back here.

You build C# programs from these core building blocks: namespaces organize your types, types (classes, structs, interfaces, enums, and delegates) define behavior and data, and statements and expressions perform work at run time. The way you structure the entry point depends on which application style you choose.

Choosing your application style

When you create a C# program, make two independent choices about how to structure it:

- **File-based or project-based?**
  - A file-based app runs from a single `.cs` file with no project file.
  - A project-based app uses a `.csproj` file and can span multiple source files.
- **Top-level statements or `Main` method?**
  - Top-level statements let you write executable code directly at the top of a file.
  - A `Main` method wraps the entry point in an explicit static method.

Both project-based apps and file-based apps support either entry-point style.

File-based apps vs. project-based apps

Starting with C# 14 and .NET 10, *file-based apps* let you run a program contained in a single `*.cs` file without a project file. Store the following code in a file named `hello-world.cs` and run it with `dotnet run hello-world.cs` or `dotnet hello-world.cs`:

C#

```
#!/usr/bin/env dotnet
Console.WriteLine("Hello, World!");
```

> **Note**

The `#!` line enables Unix operating systems to execute the file directly (for example, `./hello-world.cs`). This requires the *execute* permission to be set (`chmod +x <file>`).

File-based apps support all C# syntax and can use preprocessor directives to configure the build system. Use file-based apps for small command-line utilities, prototypes, and experiments. A file-based app consists of a single file in a directory:

```
my-app/
└── hello-world.cs
```

Project-based apps use a `.csproj` file and the .NET CLI commands `dotnet new`, `dotnet build`, and `dotnet run` workflow. Choose project-based apps when your program spans multiple files or needs fine-grained build configuration. A project-based app includes a project file alongside one or more source files:

```
my-app/
├── my-app.csproj
├── Program.cs
├── Models/
│   └── Person.cs
└── Services/
    └── GreetingService.cs
```

If your file-based app grows, you can easily convert it to a project-based app. Run dotnet project convert to generate a project file from your existing source file.

If you know your app needs multiple source files from the start, begin with a project-based app. You avoid the conversion step and can organize your code into separate files right away.

Top-level statements vs. `Main` method

By using top-level statements, you can write executable code directly in one file without wrapping it in a class and `Main` method. This style is the default when you create a new console app with `dotnet new console`. The following example shows a modern C# program that uses top-level statements:

C#

```
Console.WriteLine("Hello, World!");
```

```
namespace YourNamespace
{
    class YourClass
    {
    }

    struct YourStruct
    {
    }

    interface IYourInterface
    {
    }

    delegate int YourDelegate();

    enum YourEnum
    {
    }
}
```

Only one file in a project can have top-level statements, and the entry point is the first line of program text in that file. As you build larger programs, you include more program elements.

You can also define an explicit static Main method as the program's entry point:

C#

```
// A skeleton of a C# program using an explicit Main method
namespace YourNamespace;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello, World!");
    }
}
```

Both entry-point styles work with file-based and project-based apps. Both styles support the same features.

Building and running C# programs

C# is a *compiled* language. For project-based apps, use the dotnet build command to compile source files into a binary package. Use dotnet run to build and run in one step. The `dotnet` CLI, included in the .NET SDK, provides many tools to create, build, and manage C# projects.

For file-based apps, `dotnet run hello-world.cs` compiles and runs the single file directly - no project file required.

Expressions and statements

If you followed the Get started tutorials, you already wrote expressions and statements. Every line of code you typed was one or the other (or both). Now let's define those terms.

Expressions and statements are the fundamental building blocks of a C# program. An *expression* produces a value. A *statement* performs an action and typically ends in a semicolon.

The following are expressions:

- `42` (literal value)
- `x + y` (arithmetic operation)
- `Math.Max(a, b)` (method call that produces a value)
- `condition ? trueValue : falseValue` (conditional expression)
- `new Person("John")` (object creation)

A *statement* performs an action. Statements control program flow, declare variables, or invoke operations. The following are statements:

- `int x;` (declaration statement)
- `int x = 42;` (declaration statement with initialization)
- `Console.WriteLine("Hello");` (method call statement)
- `if (condition) { /* code */ }` (conditional statement)
- `return result;` (return statement)

Statements often contain expressions, and expressions can nest inside other expressions. For example, the following declaration statement assigns `f` to the result of an addition expression. That addition expression adds the results of two method call expressions:

C#

```
var maxResult = Math.Max(a, b) + Math.Max(c, d);
```

For detailed information about statements, see Statements. For information about expression-bodied members, see Expression-bodied members.

Related content

- Classes

- Structs
- Namespaces
- Interfaces
- Enums
- Delegates
- Basic concepts in the C# language specification

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/20/2026

Namespaces and using directives

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. They introduce namespaces and `using` directives as you write your first programs.
>
> **Experienced in another language?** Namespaces in C# work similarly to packages in Java or modules in Python. Skim ahead to the syntax you need.

Namespace declarations and `using` directives are related language features. A namespace declaration puts your types into an organized structure. A namespace groups related types together and prevents naming collisions. A `using` directive lets your program consume those types by their simple names. You don't have to spell out the full namespace path at every use.

You've already used namespaces in every C# program you've written. Every .NET type belongs to a namespace, and every `using` directive at the top of a file references one. For example, `Console` and `Math` belong to the `System` namespace, so their fully qualified names are `System.Console` and `System.Math`. Collection types like `List<T>` and `Dictionary<TKey, TValue>` belong to `System.Collections.Generic`. A single `using` directive for any of these namespaces lets you refer to all its types by their simple names. You write `List<T>` instead of `System.Collections.Generic.List<T>` everywhere you use it.

This article provides more background on how namespaces and `using` directives work, and shows examples of patterns you've already encountered in .NET libraries.

A namespace contains types. Every .NET type belongs to a namespace. For example, consider `System.Threading.Tasks.Task`: the type `Task` belongs to the `System.Threading.Tasks` namespace.

It's good practice to group related or similar types in the same namespace, and that's what .NET does with the types it provides. The `System.Collections.Generic` namespace contains collection-related types and the `System.IO` namespace contains types related reading and writing files, directories, and data. The `System` namespace contains fundamental types like `Math`, `DateTime`, and `Console`.

The following example shows how namespaces work together with `using` directives in a typical C# file:

```
C#
```

```
using System.Globalization;

namespace MyApp.Services;

class Greeter
{
    public string Greet(string name)
    {
        var culture = CultureInfo.CurrentCulture;
        return $"Hello, {name}! Culture: {culture.Name}";
    }
}
```

In the preceding sample, the `using` directive means you can use the System.Globalization.CultureInfo by the name `CultureInfo` without specifying the full name of `System.Globalization.CultureInfo`. The `namespace` directive declares that the `Greeter` class is part of the `MyApp.Services` namespace. Its fully qualified name is `MyApp.Services.Greeter`.

Namespace declarations

A namespace declaration assigns your types to a named group. Every type you write should belong to a namespace. The namespace name typically mirrors the folder structure of your project. For example, types in a `Services/Payments` folder often belong to the `MyApp.Services.Payments` namespace.

Namespaces use the `.` operator to express hierarchy, such as `System.Collections.Generic`. Namespace names must be valid C# identifier names.

File-scoped namespaces

Use the *file-scoped* syntax when all types in a file belong to the same namespace. Add a semicolon after the namespace declaration, and it applies to the entire file. You don't need extra braces or indentation:

C#

```
namespace MyApp.Models;

class Customer
{
    public required string Name { get; init; }
    public string? Email { get; init; }

    public override string ToString() => $"{Name} ({Email ?? "no email"})";
}
```

File-scoped namespaces reduce nesting and make files easier to read. You can only have one file-scoped namespace declaration per file.

> **Tip**
>
> Use file-scoped namespaces in new code. Most .NET templates and code analyzers recommend this style.

Block-scoped namespaces

Use *block-scoped* syntax when you need to declare more than one namespace in the same file. This style adds an extra level of indentation.

> **Important**
>
> It's considered bad-practice to declare more than one namespace in the same file. The more common scenario is to use *file-scoped* namespaces.

The following snippet is an example of a *block-scoped* namespace:

C#

```
namespace MyApp.Models
{
    class Product
    {
        public required string Name { get; init; }
        public decimal Price { get; init; }

        public override string ToString() => $"{Name}: {Price:C}";
    }
}
```

Using directives

Without a `using` directive, you must refer to every type by its *fully qualified name*, the complete namespace path plus the type name:

C#

```
static void ShowFullyQualified()
{
    // Without a using directive, use the fully qualified name:
```

```
    System.Console.WriteLine("Hello from fully qualified name!");
}
```

A `using` directive at the top of a file imports a namespace so you can use its types by their simple names:

C#

```
static void ShowShortName()
{
    // With 'using System;' (or implicit usings enabled), use the short name:
    Console.WriteLine("Hello from short name!");
}
```

For more information, see the using directive.

Global using directives

If you write the same `using` directives in every file, *global using* directives let you declare them once for your entire project. Place them in any file. Many teams create a dedicated `GlobalUsings.cs` file:

C#

```
global using System.Text;
global using System.Text.Json;
```

After declaring a global using, every file in the project can refer to types from that namespace by using simple names without an additional `using` directive.

Implicit usings

The .NET SDK automatically generates global using directives for the most common namespaces based on your project type. Enable implicit usings by setting `<ImplicitUsings>enable</ImplicitUsings>` in your project file. For example, a console app project automatically imports `System`, `System.Collections.Generic`, `System.IO`, `System.Linq`, `System.Threading`, and `System.Threading.Tasks`. The current SDK enables `ImplicitUsings` when you create a new project by using `dotnet new`.

For more information, see Implicit using directives.

Static using directives

A `static using` directive imports the static members of a type so you can call them without the type name prefix:

C#

```
using static System.Math;

namespace MyApp.Utilities;

class CircleCalculator
{
    public static double CalculateArea(double radius) => PI * Pow(radius, 2);

    public static double CalculateCircumference(double radius) => 2 * PI * radius;
}
```

Static usings work well for utility classes like Math and Console that you call frequently.

Type and namespace aliases

A `using` alias creates a shorthand name for a type or namespace. Aliases are useful for long generic types, resolving naming conflicts, and improving readability:

C#

```
using CustomerList = System.Collections.Generic.List<MyApp.Models.Customer>;

namespace MyApp.Services;

class CustomerService
{
    public CustomerList GetTopCustomers()
    {
        CustomerList customers = [new() { Name = "Alice" }, new() { Name = "Bob"
}];
        return customers;
    }
}
```

Starting with C# 12, you can alias any type, including tuples and pointer types:

C#

```
using Point = (double X, double Y);

namespace MyApp.Geometry;

class Shape
{
```

```
    public static double Distance(Point a, Point b)
    {
        var dx = a.X - b.X;
        var dy = a.Y - b.Y;
        return Math.Sqrt(dx * dx + dy * dy);
    }
}
```

For more advanced scenarios where two assemblies define the same fully qualified type name, use extern alias to disambiguate between them.

Related content

- using directive (language reference)
- Namespaces in the C# language specification

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/20/2026

Preprocessor directives

> **Tip**
>
> **New to developing software?** You won't need preprocessor directives right away. Focus on the Get started tutorials first and come back here when your projects require conditional compilation or build configuration.
>
> **Experienced in another language?** If you're familiar with `#ifdef` in C/C++ or conditional compilation in other languages, C# preprocessor directives work similarly. Skim ahead to the syntax you need.

C# preprocessor directives tell the compiler what code to include, exclude, or treat differently when it builds your app. This guidance can change the resulting program. Preprocessor directives always start with `#` and must appear on their own line (ignoring leading whitespace). You can add a trailing comment after the directive. While the language reference documents all available directives, three groups cover everyday use:

- **File-based apps** (`#:`) - configure file-based apps.
- **Conditional compilation** (`#if` / `#elif` / `#else` / `#endif`) — include or exclude code based on build configuration or target framework.
- **Warning suppression** (`#pragma warning`) — suppress or restore specific compiler warnings.

File-based app directives

Starting with C# 14, file-based apps use two additional directives:

- `#!` — the *shebang* line that enables executing the file directly on Unix (for example, `./Program.cs`). This requires the *execute* permission to be set on the file (`chmod +x <file>`).
- `#:` — build-system directives that configure packages, SDK settings, and other options for single-file programs.

Use `#:package` to add a NuGet package. For example, the following file-based app uses the `Spectre.Console` package to render styled output:

C#

```
#!/usr/bin/env dotnet
#:package Spectre.Console@*
```

```
AnsiConsole.MarkupLine("[bold green]Hello[/] from a file-based app!");
```

You can specify an exact version with `@`, or use `@*` to pull the latest version. Add multiple `#:package` directives to include more packages:

C#

```
#:package Serilog@3.1.1
```

Other `#:` directives let you reference projects, set MSBuild properties, or change the SDK:

C#

```
#:project ../SharedLibrary/SharedLibrary.csproj
#:property PublishAot=false
#:sdk Microsoft.NET.Sdk.Web
```

For the full list of directives, see File-based apps and the language reference.

Conditional compilation

Use `#if`, `#elif`, `#else`, and `#endif` to include or exclude code based on whether a symbol is defined. The most common symbols are `DEBUG` (set automatically for Debug builds) and target framework symbols like `NET10_0_OR_GREATER`:

C#

```
static void ConfigureLogging()
{
#if DEBUG
    Console.WriteLine("Debug logging enabled — verbose output active.");
#else
    Console.WriteLine("Release logging — errors only.");
#endif
}
```

The build system defines the `DEBUG` symbol when you build in the Debug configuration. You don't need to define it yourself. Target framework symbols like `NET10_0_OR_GREATER` and `NET8_0_OR_GREATER` let you write code that adapts to different .NET versions in multi-targeting projects.

You can combine symbols with logical operators: `&&` (and), `||` (or), and `!` (not):

C#

```
static void ShowPlatformInfo()
{
#if NET10_0_OR_GREATER
    Console.WriteLine("Running on .NET 10 or later.");
#elif NET8_0_OR_GREATER
    Console.WriteLine("Running on .NET 8 or 9.");
#else
    Console.WriteLine("Running on an older .NET version.");
#endif
}
```

Use `#define` at the top of a file to define your own symbols. You can also define symbols for the entire project by using the DefineConstants property in your project file.

Warning suppression

Use `#pragma warning disable` to suppress specific compiler warnings, and `#pragma warning restore` to re-enable them. Always scope the suppression as narrowly as possible:

C#

```
    static void ProcessData()
    {
        try
        {
            // Some operation that might fail
            var data = File.ReadAllText("config.json");
            Console.WriteLine($"Config loaded: {data.Length} characters");
        }
#pragma warning disable CS0168 // Variable is declared but never used
        catch (FileNotFoundException ex)
#pragma warning restore CS0168
        {
            // Fall back to defaults — the exception details aren't needed here
            Console.WriteLine("Config file not found, using defaults.");
        }
    }
```

> **Tip**
>
> Always specify the warning number, such as `CS0168`, rather than disabling all warnings. This approach keeps the suppression targeted and makes it clear *why* a warning is being suppressed.

Related content

- C# preprocessor directives (language reference)
- File-based apps

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/25/2026

Program organization

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You'll learn about program organization naturally as your projects grow.
>
> **Experienced in another language?** If you're familiar with solutions and projects in Visual Studio, or build systems like Maven or Cargo, this article maps those concepts to .NET.

As a C# application grows, you need to organize the code. .NET provides a hierarchy of organizational tools—solutions, projects, assemblies, namespaces, and types—that work together to keep large codebases manageable. The conventions described here represent broad consensus in the .NET community. You might deviate for specific reasons, but following these conventions makes your code familiar and navigable to other .NET developers.

The organizational hierarchy

Organize a typical .NET application in layers, from broadest to most specific:

Expand table

Level	Description	Example
Solution	A container that groups related projects.	`MyApp.slnx`
Project	A build unit that produces one assembly.	`MyApp.Web.csproj`
Assembly	The compiled `.dll` or `.exe` produced by a project.	`MyApp.Web.dll`
Namespace	A logical grouping of types.	`MyApp.Web.Controllers`
Type	A class, struct, interface, enum, or delegate.	`OrderController`

Each level serves a different purpose. Solutions organize your development workflow. Projects define what gets compiled together, and each project produces one assembly. Assemblies are the unit of deployment and versioning. Namespaces prevent naming collisions and make types easy to find. A single assembly can contain multiple namespaces, and a single namespace can span multiple assemblies. Types define the actual behavior and data.

Projects and assemblies

Each project compiles into a single assembly: a class library or executable. Start with a single project for small applications—don't split prematurely. The primary reason to create a separate project is to reuse that code in more than one application. Beyond reuse, add projects when you have a concrete reason:

- **Share code across applications** — extract shared logic into a class library that multiple apps reference.
- **Separate concerns** — keep your data access, business logic, and presentation layers independent.
- **Control dependencies** — a project can only use types from projects it explicitly references.

A single project works well for many applications. Resist the urge to create separate projects "just in case." You can always extract a library later when a second application needs the same code.

Match namespaces to folder structure

Namespace names should follow the folder structure of your project. When you see the namespace `MyApp.Services.Payments`, you know to look in the `Services/Payments` folder for the source code of the types defined in that namespace. The .NET SDK supports this convention, and it is so widely followed that violating it actively confuses other developers:

C#

```
// File: Services/OrderService.cs
// Namespace mirrors the folder path
using MyApp.Core;

namespace MyApp.Services;

public class OrderService
{
    public Order CreateOrder(string product, int quantity, decimal price) =>
        new() { ProductName = product, Quantity = quantity, UnitPrice = price };

    public string FormatSummary(Order order) =>
        $"{order.Quantity}x {order.ProductName} = {order.Total:C}";
}
```

Your root namespace is automatically set to the name of your project file. Types in subfolders don't automatically get sub-namespaces—you declare the namespace explicitly in each file—but always keep them in sync.

> **Tip**
>
> You can change the root namespace by setting `<RootNamespace>` in your project file.
>
> XML
>
> ```xml
> <Project Sdk="Microsoft.NET.Sdk">
>  <PropertyGroup>
>    <RootNamespace>MyCompany.MyApp</RootNamespace>
>  </PropertyGroup>
> </Project>
> ```

Organize namespaces by feature, not by type kind

Group related types into namespaces by feature or responsibility. Place an interface, its implementations, and supporting types together:

C#

```csharp
// Good: group by feature
namespace MyApp.Payments;

public interface IPaymentProcessor
{
    bool ProcessPayment(decimal amount);
}

public class CreditCardProcessor : IPaymentProcessor
{
    public bool ProcessPayment(decimal amount)
    {
        Console.WriteLine($"Processing credit card payment of {amount:C}");
        return true;
    }
}

public record PaymentResult(bool Success, string? TransactionId);
```

Feature-based organization keeps everything you need in one place, making the code easier to navigate and reason about.

Access modifiers and assemblies

Access modifiers work with the project and assembly structure to control accessibility:

- public — accessible from any assembly that references this assembly.
- internal — accessible only within the same assembly (the default for top-level types).
- private, protected, private protected, protected internal — accessible based on the containing type, the assembly, or derived types.

Default to `internal` for types that other projects don't need. This practice hides implementation details and gives you freedom to refactor without breaking consumers. It's especially important for shared libraries:

C#

```
namespace MyApp.Inventory;

// Public — other projects can use this type
public class InventoryService
{
    public int GetStockLevel(string productName) =>
        StockDatabase.Lookup(productName);
}

// Internal — only visible within this assembly
internal static class StockDatabase
{
    private static readonly Dictionary<string, int> _stock = new()
    {
        ["Widget"] = 42,
        ["Gadget"] = 17
    };

    internal static int Lookup(string productName) =>
        _stock.GetValueOrDefault(productName);
}
```

Recommended practices

- **Name namespaces consistently.** Use `CompanyName.ProductName.Feature` as your naming pattern. For example, use `Contoso.Inventory.Shipping`. Consistent naming helps developers find types without searching.
- **Keep projects focused.** Each project should have a single, clear responsibility. When a project handles too many unrelated concerns, split it.
- **Use file-scoped namespaces.** The `namespace MyApp.Services;` syntax reduces indentation and is the recommended style. Use it in all new code.
- **Default to `internal`.** Only mark types `public` when other assemblies genuinely need them. You can always widen access later; narrowing it is a breaking change.

Related content

- Namespaces and using directives
- Assemblies in .NET
- .NET project SDKs

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/20/2026

Top-level statements - programs without `Main` methods

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. Those tutorials use top-level statements, so you'll already be familiar with the basics.
>
> **Looking for the `Main` method alternative?** See Main method entry point for the explicit `Main` method approach.

Use *top-level statements* for new apps. By using top-level statements, you can write executable code directly at the root of a file.

Here's a *Program.cs* file that is a complete C# program:

```
C#

Console.WriteLine("Hello World!");
```

When you create a new console app by using `dotnet new console`, it uses top-level statements by default. They work well for programs of any size - from small utilities and Azure Functions to full applications. If you have an existing application that uses an explicit `Main` method, there's no need to convert it. Both styles compile to equivalent code.

The following sections explain the rules on what you can and can't do with top-level statements.

Entry point rules

An application must have only one entry point. A project can have only one file with top-level statements, but it can have any number of source code files that don't have top-level statements. You can explicitly write a `Main` method, but it can't function as an entry point. In a project with top-level statements, you can't use the -main compiler option to select the entry point, even if the project has one or more `Main` methods.

The compiler generates a method to serve as the program entry point for a project with top-level statements. The signature of the method depends on whether the top-level statements contain the `await` keyword or the `return` statement. The following table shows what the method signature looks like, using the method name `Main` in the table for convenience.

Expand table

Top-level code contains	Implicit `Main` signature
`await` and `return`	`static async Task<int> Main(string[] args)`
`await`	`static async Task Main(string[] args)`
`return`	`static int Main(string[] args)`
No `await` or `return`	`static void Main(string[] args)`

Starting with C# 14, programs can be *file-based apps*, where a single file contains the program. You run *file-based apps* by using the command `dotnet <file.cs>`, or directly using the filename on Unix (for example, `./file.cs`). The latter requires including the `#!/usr/bin/env dotnet` directive as the first line and setting the execute permission (`chmod +x <file>`).

`using` directives

For the single file containing top-level statements, `using` directives must come first in that file, as in the following example:

C#

```
using System.Text;

StringBuilder builder = new();
builder.AppendLine("The following arguments are passed:");

foreach (var arg in args)
{
    builder.AppendLine($"Argument={arg}");
}

Console.WriteLine(builder.ToString());

return 0;
```

Namespaces and type definitions

Top-level statements are implicitly in the global namespace. A file with top-level statements can also contain namespaces and type definitions, but they must come after the top-level statements. For example:

C#

```
MyClass.TestMethod();
MyNamespace.MyClass.MyMethod();

public class MyClass
{
    public static void TestMethod()
    {
        Console.WriteLine("Hello World!");
    }
}

namespace MyNamespace
{
    class MyClass
    {
        public static void MyMethod()
        {
            Console.WriteLine("Hello World from MyNamespace.MyClass.MyMethod!");
        }
    }
}
```

`args`

Top-level statements can reference the `args` variable to access any command-line arguments passed to the app when it starts. The `args` variable is never `null`, but its `Length` is zero if no command-line arguments were provided. For example:

C#

```
if (args.Length > 0)
{
    foreach (var arg in args)
    {
        Console.WriteLine($"Argument={arg}");
    }
}
else
{
    Console.WriteLine("No arguments");
}
```

`await` and exit code

Use `await` to call an async method. When your top-level code contains `await`, the compiler generates an entry point that returns a `Task`. The runtime monitors that `Task` for completion, keeping the process alive until all asynchronous work finishes. For example:

C#

```
Console.Write("Hello ");
await Task.Delay(5000);
Console.WriteLine("World!");
```

To return an exit code when the application ends, use the `return` statement. The compiler generates an entry point that returns `Task<int>` when your code contains both `await` and `return`, or `int` when it contains only `return`. For example:

C#

```
string? s = Console.ReadLine();

int returnValue = int.Parse(s ?? "-1");
return returnValue;
```

Related content

- Main() and command-line arguments
- General structure of a C# program
- Feature specification - Top-level statements

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/25/2026

Main() and command-line arguments

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. Those tutorials use top-level statements, which is simpler for new apps.
>
> **Working with an existing codebase?** Many existing applications use an explicit `Main` method. This article explains how it works and how to use it effectively.

When you start a C# application, the runtime calls the `Main` method. The `Main` method is the entry point of a C# application.

A C# program can have only one entry point. If you have more than one class with a `Main` method, you must use the **StartupObject** compiler option when you compile your program to specify which `Main` method serves as the entry point. For more information, see StartupObject (C# Compiler Options). The following example displays the number of command-line arguments as its first action:

C#

```
class TestClass
{
    static void Main(string[] args)
    {
        Console.WriteLine(args.Length);
    }
}
```

Overview

The `Main` method is the entry point of an executable program. When your program starts, the runtime calls `Main` before any other code runs. When `Main` returns, the program ends. You declare `Main` with these rules:

- You must declare `Main` inside a class or struct. The enclosing `class` can be `static`.
- `Main` must be static.
- `Main` can have any access modifier.
- `Main` can return `void`, `int`, `Task`, or `Task<int>`.
- If and only if `Main` returns a `Task` or `Task<int>`, the declaration of `Main` can include the async modifier. This rule specifically excludes an `async void Main` method.

- You can declare the `Main` method with or without a `string[]` parameter that contains command-line arguments. When you use Visual Studio to create Windows applications, you can add the parameter manually or else use the GetCommandLineArgs() method to obtain the command-line arguments. Parameters are zero-indexed command-line arguments. Unlike C and C++, the name of the program isn't treated as the first command-line argument in the `args` array, but it's the first element of the GetCommandLineArgs() method.

The following list shows permutations of `Main` declarations:

C#

```
static void Main() { }
static int Main() { }
static void Main(string[] args) { }
static int Main(string[] args) { }
static async Task Main() { }
static async Task<int> Main() { }
static async Task Main(string[] args) { }
static async Task<int> Main(string[] args) { }
```

The preceding examples don't specify an access modifier, so they're implicitly `private` by default. You can specify any explicit access modifier.

The following table summarizes all valid `Main` signatures and when to use each one:

Expand table

`Main` declaration	Uses `args`	contains `await`	Returns exit code
`static void Main()`	No	No	No
`static int Main()`	No	No	Yes
`static void Main(string[] args)`	Yes	No	No
`static int Main(string[] args)`	Yes	No	Yes
`static async Task Main()`	No	Yes	No
`static async Task<int> Main()`	No	Yes	Yes
`static async Task Main(string[] args)`	Yes	Yes	No
`static async Task<int> Main(string[] args)`	Yes	Yes	Yes

Choose the simplest signature that fits your needs. If you don't need command-line arguments, omit the `string[] args` parameter. If you don't need to return an exit code, use `void` or `Task`. If you need to call asynchronous methods, use `async` with a `Task` or `Task<int>` return type.

Main() return values

When you return `int` or `Task<int>`, your program can send status information to other programs or scripts that run the executable. A return value of `0` usually means success, and a nonzero value means there's an error.

The following example returns an exit code:

C#

```
class MainReturnValTest
{
    static int Main()
    {
        //...
        return 0;
    }
}
```

After running the program, you can check the exit code. In PowerShell, use `$LastExitCode`. In a batch file or shell script, use `%ERRORLEVEL%`.

If your `Main` method uses `await`, declare it as `async` with a `Task` or `Task<int>` return type. The runtime calls `Main` and waits for the returned `Task` to complete before the process exits. The return type can't be `void` or `int` because the `async` modifier requires a return type that the runtime can await—`void` and `int` don't represent ongoing work, so the process could exit before asynchronous operations finish. Use `Task` when you don't need an exit code, or `Task<int>` when you do:

C#

```
class Program
{
    static async Task<int> Main(string[] args)
    {
        return await AsyncConsoleWork();
    }

    private static async Task<int> AsyncConsoleWork()
    {
```

```
        return 0;
    }
}
```

Command-line arguments

Include a `string[] args` parameter in your `Main` declaration to accept command-line arguments. If you don't need them, omit the parameter. The `args` parameter is a String array that's never null—if no arguments are provided, its `Length` is zero.

You can convert string arguments to other types by using `Parse` or Convert:

C#

```
long num = long.Parse(args[0]);
```

> **Tip**
>
> Parsing command-line arguments can be complex. Consider using the System.CommandLine library to simplify the process.

For a working example, see How to display command-line arguments.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Related content

- System.Environment
- How to display command line arguments

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/20/2026

The C# type system

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. They walk you through writing programs and introduce types as you go.
>
> **Experienced in another language?** If you already understand type systems, skim the value vs. reference distinction and the choose which kind of type guide, then jump to the articles on specific types.

C# is a strongly typed language. Every variable, constant, and expression has a type. The compiler enforces *type safety* by checking that every operation in your code is valid for the types involved. For example, you can add two `int` values, but you can't add an `int` and a `bool`:

C#

```
int a = 5;
int b = a + 2; // OK

bool test = true;

// Error. Operator '+' cannot be applied to operands of type 'int' and 'bool'.
// int c = a + test;
```

> **Note**
>
> Unlike C and C++, in C#, `bool` isn't convertible to `int`.

Type safety catches errors at compile time, before your code runs. The compiler also embeds type information into the executable as metadata, which the common language runtime (CLR) uses for additional safety checks at run time.

Declare variables with types

When you declare a variable, you specify its type explicitly or use `var` to let the compiler infer the type from the assigned value:

C#

```
// Explicit type:
int count = 10;
double temperature = 36.6;

// Compiler-inferred type:
var name = "C#";
var items = new List<string> { "one", "two", "three" };
```

Method parameters and return values also have types. The following method takes a `string` and an `int`, and returns a `string`:

C#

```
static string GetGreeting(string name, int visitCount)
{
    return visitCount switch
    {
        1 => $"Welcome, {name}!",
        _ => $"Welcome back, {name}! Visit #{visitCount}."
    };
}
```

After you declare a variable, you can't change its type or assign a value that's incompatible with the declared type. You can convert values to other types. The compiler performs *implicit conversions* that don't lose data automatically. *Explicit conversions* (casts) require you to indicate the conversion in your code. For more information, see Casting and type conversions.

Built-in types and custom types

C# provides built-in types for common data: integers, floating-point numbers, `bool`, `char`, and `string`. Every C# program can use these built-in types without any extra references.

Beyond built-in types, you can create your own types by using several constructs:

- **Classes** — Reference types for modeling behavior and complex objects. Support inheritance and polymorphism.
- **Structs** — Value types for small, lightweight data. Each variable holds its own copy.
- **Records** — Classes or structs with compiler-generated equality, `ToString`, and nondestructive mutation through `with` expressions.
- **Interfaces** — Contracts that define members any class or struct can implement.
- **Enumerations** — Named sets of integral constants, such as days of the week or file access modes.
- **Tuples** — Lightweight structural types that group related values without defining a named type.

- **Generics** — Type-parameterized constructs like `List<T>` and `Dictionary<TKey, TValue>` that provide type safety while reusing the same logic for different types.

Value types and reference types

Every type in C# is either a *value type* or a *reference type*. This distinction determines how variables store data and how assignment works.

Value types hold their data directly. When you assign a value type to a new variable, the runtime copies the data. Changes to one variable don't affect the other. Structs, enums, and the built-in numeric types are all value types.

Reference types hold a reference to an object on the managed heap. When you assign a reference type to a new variable, both variables point to the same object. Changes through one variable are visible through the other. Classes, arrays, delegates, and strings are reference types.

The following example shows the difference. The first block shows the definition for the `Coords` record struct, which is a value type. The second block shows the different behavior for value types and reference types.

C#

```
public readonly record struct Coords(int X, int Y);
```

C#

```
// Value type: each variable holds its own copy
var point1 = new Coords(3, 4);
var point2 = point1;
Console.WriteLine($"point1: ({point1.X}, {point1.Y})");
Console.WriteLine($"point2: ({point2.X}, {point2.Y})");
// point1 and point2 are independent copies

// Reference type: both variables refer to the same object
var list1 = new List<int> { 1, 2, 3 };
var list2 = list1;
list2.Add(4);
Console.WriteLine($"list1 count: {list1.Count}"); // 4 — same object
```

All types ultimately derive from System.Object. Value types derive from System.ValueType, which derives from `object`. This unified hierarchy is called the Common Type System (CTS). For more information about inheritance, see Inheritance.

Choose which kind of type

When you define a new type, the kind you choose shapes how your code behaves. Use the following guidelines to make an initial decision:

- **Tuple** — Temporary grouping of values that doesn't need a named type or behavior.
- `struct` or `record struct` — Small data (roughly 64 bytes or less), value semantics, or immutability. Record structs add value-based equality and `with` expressions.
- `record class` — Primarily data, with value-based equality, `ToString`, and nondestructive mutation. Supports inheritance.
- `class` — Complex behavior, polymorphism, or mutable state. Most custom types are classes.
- `interface` — A contract that unrelated types can implement. Defines capabilities rather than identity.
- `enum` — A fixed set of named constants, such as status codes or options.

More than one option is often reasonable.

Compile-time type and run-time type

A variable can have different types at compile time and run time. The *compile-time type* is the declared or inferred type in source code. The *run-time type* is the actual type of the instance the variable refers to. The run-time type must be the same as the compile-time type, or a type that derives from it or implements it. An assignment is only valid when an implicit conversion exists from the run-time type to the compile-time type, such as an identity, reference, boxing, or numeric conversion.

C#

```
// Compile-time and run-time types match:
string message = "Hello, world!";

// Compile-time type differs from run-time type:
object boxed = "This is a string at run time";
IEnumerable<char> characters = "abcdefghijklmnopqrstuvwxyz";
```

In the preceding example, `boxed` has a compile-time type of `object` but a run-time type of `string`. The assignment works because `string` derives from `object`. Similarly, `characters` has a compile-time type of `IEnumerable<char>`, and the assignment works because `string` implements that interface. The compile-time type controls overload resolution and available conversions. The run-time type controls virtual method dispatch, `is` expressions, and `switch` expressions.

See also

- Value types
- Reference types
- Common Type System

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/26/2026

Built-in types and literals

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. They introduce types as you write your first programs.
>
> **Experienced in another language?** C# built-in types map closely to types in Java, C++, and other languages. Skim the literal syntax and type inference sections for C#-specific details.

C# provides a set of built-in types that you can use in any program without additional references. These types cover the most common data you work with: numbers, true/false values, individual characters, and text. For a complete reference table mapping C# keywords to .NET types, see Built-in types (C# reference).

Numeric types

C# has built-in types for integers, floating-point numbers, and decimal numbers. The most commonly used types are `int`, `double`, and `decimal`:

C#

```
int population = 67_000_000;
long distance = 384_400_000L;
short temperature = -40;
byte red = 255;

double pi = 3.141592653589793;
float gravity = 9.81f;
decimal price = 19.99m;
```

Each numeric type has a fixed size and range. `int` stores 32-bit integers (roughly ±2.1 billion), `long` stores 64-bit integers, and `short` and `byte` store smaller values. For the full list of sizes and ranges, see Integral numeric types and Floating-point numeric types.

Use `double` for general floating-point math, `float` when memory is constrained, and `decimal` when you need exact decimal precision (such as financial calculations). Append the `f` suffix for `float` literals and `m` for `decimal` literals. Without a suffix, the compiler treats a number with a decimal point as `double`.

Unsigned types

Each signed integer type has an unsigned counterpart that stores only non-negative values with twice the positive range:

```C#
uint fileSize = 4_294_967_295;
ulong totalBytes = 18_446_744_073_709_551_615;
ushort port = 443;
```

You might use unsigned types when negative values aren't valid for the data, such as file sizes or network ports. In practice, many applications use `int` or `long` even for positive-only values because signed types are the default throughout the .NET APIs.

Native-sized integers

The `nint` and `nuint` types represent integers whose size matches the platform's native pointer size: 32 bits on a 32-bit platform, 64 bits on a 64-bit platform:

```C#
nint bufferSize = 1024;
nuint elementCount = 256;
```

You're unlikely to need `nint` or `nuint` in everyday code. They exist for interop scenarios and low-level memory operations where matching the platform's pointer size is important. Stick with `int` or `long` unless you have a specific reason to use native-sized types. For more information, see nint and nuint.

`bool`, `char`, and `string`

Beyond numbers, C# provides three other frequently used built-in types:

```C#
bool isValid = true;
char grade = 'A';
string greeting = "Hello, world!";
```

- **`bool`** — Stores `true` or `false`. Use it in conditions, loops, and logical expressions.
- **`char`** — Stores a single Unicode character (UTF-16 code unit), enclosed in single quotes.
- **`string`** — Stores a sequence of characters, enclosed in double quotes. Strings are *immutable*. Once you create a string, you can't change its contents. Operations that

appear to modify a string actually create a new one.

Strings are one of the most used types in C#. For in-depth coverage of string operations, including interpolation, raw string literals, searching, splitting, and comparison, see the Strings section.

Literal syntax

A *literal* is a value you write directly in your code. The compiler assigns each literal a type based on its format and any suffix you provide. C# supports the following kinds of literals:

- **Integer literals** — Decimal (`42`), hexadecimal (`0x2A`), and binary (`0b_0010_1010`).
- **Floating-point literals** — `double` by default (`3.14`), `float` with the `f` suffix (`3.14f`), and `decimal` with `m` (`3.14m`).
- **Character literals** — A single character in single quotes (`'A'`), including escape sequences (`'\n'`).
- **String literals** — Regular (`"hello"`), verbatim (`@"C:\path"`), raw (`""" ... """`), and interpolated (`$"value: {x}"`).
- **Boolean literals** — `true` and `false`.
- **The `null` literal** — Represents the absence of a value for reference types and nullable value types.
- **The `default` literal** — Produces the default value for any type (covered in default expressions).

The following sections cover the most common literal forms in detail.

Integer literals

C#

```
int dec = 42;
int hex = 0x2A;
int bin = 0b_0010_1010;
long big = 1_000_000_000L;
```

Use the `0x` prefix for hexadecimal and `0b` for binary. Append `L` for `long`, `U` for `uint`, or `UL` for `ulong`.

Place the `_` digit separator anywhere within a number to make it easier to read. Common patterns include thousand separators in decimal literals (`1_000_000_000`), byte or word boundaries in hexadecimal (`0xFF_FF`), and nibble boundaries in binary (`0b_0010_1010`).

Floating-point literals

C#

```
double d = 3.14;
float f = 3.14f;
decimal m = 3.14m;
double scientific = 1.5e6; // 1,500,000
```

Without a suffix, a numeric literal with a decimal point is `double`. Append `f` for `float` and `m` for `decimal`. Scientific notation (`1.5e6`) is also supported.

Character and string literals

C#

```
char newline = '\n';
char unicode = '\u0041'; // 'A'

string message = $"Found {dec} items";            // interpolated string
string path = @"C:\Users\docs\file.txt";         // verbatim string
string json = """
    { "name": "Alice", "age": 30 }
    """;                                           // raw string literal
string raw = $"""
    Found {dec} items in "{greeting}"
    """;                                           // raw + interpolated
```

Character literals use single quotes and support escape sequences (`\n`, `\t`, `\u`). String literals use double quotes.

Prefix strings with `$` for interpolation. When a string contains quotes, backslashes, or embedded JSON or XML, use a raw string literal (delimited by `"""`) instead of escaping each character. Raw string literals also combine with interpolation (`$"""`).

Older code uses `@` (verbatim strings) to avoid escape processing. Raw string literals are easier to read and write, so prefer them for new code.

`default` expressions

The `default` expression produces the default value for a type: `0` for numeric types, `false` for `bool`, and `null` for reference types:

C#

```
int defaultInt = default;          // 0
bool defaultBool = default;        // false
string? defaultString = default;   // null

// Use default in a conditional:
var limit = (args.Length > 0) ? int.Parse(args[0]) : default(int);
```

The `default` expression is most useful in generic code, where you don't know the concrete type and can't hard-code a specific value like `0` or `null`. Write `default` (without a type argument) when the compiler can infer the type from context, or `default(T)` when the type isn't obvious. For the complete list of default values by type, see Default values of C# types.

Implicitly typed variables with `var`

The `var` keyword tells the compiler to infer a local variable's type from its initializer:

C#

```
var count = 10;              // compiler infers int
var name = "C#";             // compiler infers string
var items = new List<int>(); // compiler infers List<int>

// var requires an initializer — the compiler needs a value to infer the type.
// The following line wouldn't compile:
// var unknown;
```

The variable is still strongly typed; `var` doesn't make it dynamic. The compiler determines the type at compile time and enforces type safety as usual. Use `var` when the type is obvious from the right-hand side to reduce visual noise. Spell out the type when it makes the code clearer. For more information, see Implicitly typed local variables.

Target-typed `new` expressions

When you already know the target type from context, such as a variable declaration or a method parameter, you can omit the type name from the `new` expression:

C#

```
List<string> names = new() { "Alice", "Bob", "Charlie" };
Dictionary<string, int> scores = new()
{
    ["Alice"] = 95,
    ["Bob"] = 87
};
```

Target-typed `new` reduces repetition when the type name is long or appears on the left-hand side of the assignment. It works anywhere the compiler can determine the target type, including method arguments and return statements. For more information, see new operator — target-typed new.

The `dynamic` type

The `dynamic` type bypasses compile-time type checking. The compiler resolves operations on a `dynamic` variable at run time instead:

C#

```
dynamic value = 42;
Console.WriteLine(value.GetType()); // System.Int32

value = "Now I'm a string";
Console.WriteLine(value.GetType()); // System.String

// The compiler doesn't check operations on dynamic at compile time.
// Errors surface at run time instead.
```

Use `dynamic` when interacting with COM APIs, dynamic languages, or reflection-heavy scenarios where types aren't known at compile time. Avoid `dynamic` in most application code because you lose compile-time safety. Errors that the compiler normally catches become run-time exceptions instead. For more information, see The dynamic type.

See also

- Type system overview
- Built-in types (C# reference)
- Integral numeric types
- Floating-point numeric types
- Value types

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/26/2026

C# classes

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You'll encounter classes once you need to model objects with behavior and state.
>
> **Experienced in another language?** C# classes are similar to classes in Java or C++. Skim the object initializers section for C#-specific patterns, and see Records for a data-focused alternative.

A *class* is a reference type that defines a blueprint for objects. When you create a variable of a class type, the variable holds a *reference* to an object on the managed heap. The variable doesn't hold the object data itself. Assigning a class variable to another variable copies the reference, so both variables point to the same object. Classes are the most common way to define custom types in C#. Use them when you need complex behavior, inheritance, or shared identity between references.

When to use classes

Use a class when:

- The type has complex behavior or manages mutable state.
- You need inheritance to create a base class with derived specializations, or to create a derived type that extends an existing class.
- Instances represent a shared identity, not objects that happen to hold equal values. Two references to the same object should stay in sync.
- The type is large or long-lived and benefits from heap allocation and reference semantics.

Declare a class

Define a class with the `class` keyword followed by the type name. An optional access modifier controls visibility. The default is `internal`. Specify `public` to allow callers from other assemblies to use your types.

C#

```
public class Customer
{
    public string Name { get; set; }

    public Customer(string name) => Name = name;
}
```

The class body contains fields, properties, methods, and events, collectively called *class members*. The name must be a valid C# identifier name.

Create objects

A class defines a type, but it isn't an object itself. You create an object (an *instance* of the class) with the `new` keyword:

C#

```
var customer = new Customer("Allison");
Console.WriteLine(customer.Name); // Allison
```

The variable `customer` holds a reference to the object, not the object itself. You can assign multiple variables to the same object. Changes through one reference are visible through the other:

C#

```
var c1 = new Customer("Grace");
var c2 = c1; // both variables reference the same object

c2.Name = "Hopper";
Console.WriteLine(c1.Name); // Hopper — c1 sees the change made through c2
```

This reference-sharing behavior is one distinction between classes and structs. With structs, assignment copies the data. More importantly, classes support inheritance. You can build hierarchies where derived types reuse and specialize behavior from a base class. Structs can't participate in inheritance hierarchies. For more on the distinction, see Value types and reference types.

Constructors and initialization

When you create an instance, you want its fields and properties initialized to useful values. C# offers several approaches: field initializers, constructor parameters, primary constructors, and

required properties.

Field initializers set a default value directly on the field declaration:

C#

```
public class Container
{
    private int _capacity = 10;
}
```

Field initializers assign a reasonable default to a field or property. This distinguishes it from the following approaches where callers can provide the initial value.

Constructor parameters require callers to provide values:

C#

```
public class Container
{
    private int _capacity;

    public Container(int capacity) => _capacity = capacity;
}
```

Primary constructors (C# 12+) add parameters directly to the class declaration. Those parameters are available throughout the class body:

C#

```
public class Container(int capacity)
{
    private int _capacity = capacity;
}
```

Primary constructors and field initializers can work together: the field initializer `_capacity = capacity` uses the primary-constructor parameter as its value. This pattern lets you capture constructor arguments in fields with a single, concise declaration.

Required properties enforce that callers set specific properties through an object initializer:

C#

```
public class Person
{
    public required string FirstName { get; set; }
```

```
    public required string LastName { get; set; }
}
```

C#

```
// var missing = new Person(); // Error: required properties not set
var person = new Person { FirstName = "Grace", LastName = "Hopper" };
Console.WriteLine($"{person.FirstName} {person.LastName}"); // Grace Hopper
```

For a deeper look at constructor patterns, including parameter validation and constructor chaining, see Constructors.

Static classes

A `static` class can't be instantiated and contains only static members. Use static classes to organize utility methods that don't operate on instance data:

C#

```
static class MathHelpers
{
    public static double CircleCircumference(double radius) =>
        2 * Math.PI * radius;
}
```

C#

```
double circumference = MathHelpers.CircleCircumference(5.0);
Console.WriteLine($"Circumference: {circumference:F2}"); // Circumference: 31.42
```

The .NET class library includes many static classes, such as Math and Console. A static class is implicitly sealed. You can't derive from it or instantiate it.

Object initializers

Object initializers let you set properties when you create an object, without writing a constructor for every combination of values:

C#

```
class ConnectionOptions
{
    public string Host { get; init; } = "localhost";
    public int Port { get; init; } = 80;
```

```
    public bool UseSsl { get; init; }
}
```

C#

```
var options = new ConnectionOptions
{
    Host = "db.example.com",
    Port = 5432,
    UseSsl = true
};
Console.WriteLine($"{options.Host}:{options.Port} (SSL: {options.UseSsl})");
// db.example.com:5432 (SSL: True)
```

Object initializers work with any accessible property that has a `set` or init accessor. They combine naturally with `required` properties and with constructors that accept some parameters while letting the caller set others.

When the property is a *collection*, you can use a Collection expressions (C# reference) to initialize that object.

Inheritance

Classes support *inheritance*. You can define a new class that reuses, extends, or modifies the behavior of an existing class. The class you inherit from is the *base class*, and the new class is the *derived class*:

C#

```
var manager = new Manager("Satya", "Engineering");
Console.WriteLine($"{manager.Name} manages {manager.Department}");
// Satya manages Engineering
```

A class can inherit from one base class and implement multiple interfaces. Derived classes inherit all members of the base class except constructors. For more information, see Inheritance and Interfaces.

See also

- Type system overview
- Structs
- Records
- Interfaces

- Inheritance

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/08/2026

C# structs

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You'll encounter structs once you need lightweight value types in your code.
>
> **Experienced in another language?** C# structs are value types similar to structs in C++ or Swift, but they live on the managed heap when boxed and support interfaces, constructors, and methods. Skim the readonly structs section for C#-specific patterns. For record structs, see Records.

A *struct* is a value type that holds its data directly in the instance, rather than through a reference to an object on the heap. When you assign a struct to a new variable, the runtime copies the entire instance. Changes to one variable don't affect the other because each variable represents a different instance. Use structs for small, lightweight types whose primary role is storing data rather than modeling behavior. Examples include coordinates, colors, measurements, or configuration settings.

When to use structs

Use a struct when your type:

- Represents a single value or a small group of related values (roughly 16 bytes or less).
- Has value semantics—two instances with the same data should be equal.
- Is primarily a data container rather than a model of behavior.
- Doesn't need inheritance from a base type (structs can't inherit from other structs or classes, but they can implement interfaces).

For a broader comparison that includes classes, records, tuples, and interfaces, see Choose which kind of type.

Declare a struct

Define a struct with the `struct` keyword. A struct can contain fields, properties, methods, and constructors, just like a class:

C#

```
struct Point
{
    public double X { get; set; }
    public double Y { get; set; }

    public readonly double DistanceTo(Point other)
    {
        var dx = X - other.X;
        var dy = Y - other.Y;
        return Math.Sqrt(dx * dx + dy * dy);
    }

    public override string ToString() => $"({X}, {Y})";
}
```

The `Point` struct stores two `double` values and provides a method to calculate the distance between two points. The `DistanceTo` method is marked `readonly` because it doesn't modify the struct's state. That pattern is covered in readonly members.

Value semantics

Structs are *value types*. Assignment copies the data, so each variable holds its own independent copy:

C#

```
var p1 = new Point { X = 3, Y = 4 };
var p2 = p1; // copies the data
p2.X = 10;

Console.WriteLine(p1); // (3, 4)  — p1 is unchanged
Console.WriteLine(p2); // (10, 4) — only p2 was modified
```

Because structs are data containers, assignment copies every data member into a new, independent instance. Each copy is distinct. Modifying one doesn't affect the other. This behavior differs from classes, where assignment copies only the reference and both variables share the same object. For more on the distinction, see Value types and reference types.

Struct constructors

You can define constructors in structs the same way you do in classes. Structs can have *parameterless constructors* that set custom default values. The term "parameterless constructor"

distinguishes an instance created with `new` (which runs your constructor logic) from a *default* instance created with the `default` expression (which zero-initializes all fields):

C#

```
struct ConnectionSettings
{
    public string Host { get; set; }
    public int Port { get; set; }
    public int MaxRetries { get; set; }

    public ConnectionSettings()
    {
        Host = "localhost";
        Port = 8080;
        MaxRetries = 3;
    }
}
```

A parameterless constructor runs when you use `new` with no arguments. The `default` expression bypasses the constructor and sets all fields to their default values (`0`, `null`, `false`). Be aware of the difference:

C#

```
var custom = new ConnectionSettings();
Console.WriteLine($"{custom.Host}:{custom.Port} (retries: {custom.MaxRetries})");
// localhost:8080 (retries: 3)

var defaults = default(ConnectionSettings);
Console.WriteLine($"{defaults.Host ?? "(null)"}:{defaults.Port} (retries:
{defaults.MaxRetries})");
// (null):0 (retries: 0)
```

The compiler automatically initializes any fields you don't explicitly set in a constructor. You can initialize only the fields that need non-default values:

C#

```
struct GameTile
{
    public int Row { get; set; }
    public int Column { get; set; }
    public bool IsBlocked { get; set; }

    public GameTile(int row, int column)
    {
        Row = row;
        Column = column;
```

```
        // IsBlocked is automatically initialized to false
    }
}
```

The following example displays the default value for `IsBlocked`:

C#

```
var tile = new GameTile(2, 5);
Console.WriteLine($"Tile ({tile.Row}, {tile.Column}), blocked: {tile.IsBlocked}");
// Tile (2, 5), blocked: False
```

The `IsBlocked` property isn't assigned in the constructor, so the compiler sets it to `false` (the default for `bool`). This feature reduces boilerplate in constructors that only need to set a few fields.

Readonly structs and readonly members

A `readonly struct` guarantees that no instance member modifies the struct's state. The compiler enforces this guarantee by requiring all fields and auto-implemented properties to be read-only:

C#

```
readonly struct Temperature
{
    public double Celsius { get; }

    public Temperature(double celsius) => Celsius = celsius;

    public double Fahrenheit => Celsius * 9.0 / 5.0 + 32.0;

    public override string ToString() => $"{Celsius:F1}°C ({Fahrenheit:F1}°F)";
}
```

The following example creates a `Temperature` instance and reads its properties:

C#

```
var temp = new Temperature(100);
Console.WriteLine(temp); // 100.0°C (212.0°F)
// temp.Celsius = 50; // Error: property is read-only
```

When you don't need the entire struct to be immutable, mark individual members as `readonly` instead. A `readonly` member can't modify the struct's state, and the compiler verifies that

guarantee:

C#

```
struct Velocity
{
    public double X
    {
        readonly get;
        set;
    }

    public double Y
    {
        readonly get;
        set;
    }

    public readonly double Speed => Math.Sqrt(X * X + Y * Y);

    public readonly override string ToString() => $"({X}, {Y}) speed={Speed:F2}";
}
```

The following example shows that `readonly` members return updated values when mutable properties change:

C#

```
var v = new Velocity { X = 3, Y = 4 };
Console.WriteLine(v.Speed); // 5
Console.WriteLine(v);       // (3, 4) speed=5.00
v.X = 6;
Console.WriteLine(v.Speed); // 7.211...
```

Marking members `readonly` helps the compiler optimize defensive copies. When you pass a `readonly` struct to a method that accepts an `in` parameter, the compiler knows no copy is needed.

See also

- Type system overview
- Classes
- Records
- Structure types (C# reference)

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Last updated on 04/08/2026

C# record types

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You'll encounter records when you need concise data types with built-in equality.
>
> **Experienced in another language?** C# records are similar to data classes in Kotlin or case classes in Scala. They're types optimized for storing data, with compiler-generated equality, `ToString`, and copy semantics. Skim the record class vs record struct and with expressions sections for C#-specific patterns.

The record keyword is a modifier you apply to either a `class` or a `struct`. It tells the compiler to generate value equality, a formatted `ToString`, and nondestructive mutation through with expressions. The underlying type, a class or a struct, still determines whether instances use reference or value semantics. The `record` modifier adds data-friendly behavior on top of those semantics. Use records when a type's primary role is storing data and two instances with the same values should be considered equal.

When to use records

Use a record when all of the following conditions are true:

- The type's primary role is storing data.
- Two instances with the same values should be equal.
- You want immutability (especially for `record class` types).
- You want a readable `ToString` without writing one manually.

When choosing between `record class` and `record struct`:

- Use `record class` when you need inheritance, or when the type is large enough that copying on every assignment would be expensive.
- Use `record struct` for small, self-contained values where copy semantics and stack allocation are beneficial.

Avoid records for entity types in Entity Framework Core, which depends on reference equality to track entities. For a broader comparison of type options, see Choose which kind of type.

Declare a record

You can apply `record` to either a class or a struct. The simplest form uses *positional parameters* that define both the constructor and the properties in a single line:

C#

```
public record Person(string FirstName, string LastName);
```

The same positional syntax works for `record struct` types:

C#

```
public record struct Coordinate(double Latitude, double Longitude);

public readonly record struct Temperature(double Celsius)
{
    public double Fahrenheit => Celsius * 9.0 / 5.0 + 32.0;
}
```

Writing `record` alone is shorthand for record class which is a reference type. Writing record struct creates a value type. The compiler generates properties from the positional parameters in both cases, but the defaults differ:

- **`record class`**: Properties are `init`-only (immutable after construction).
- **`record struct`**: Properties are read-write by default. Add `readonly` (`readonly record struct`) to make them `init`-only.

You can also write records with standard property syntax when you need more control. For example, to make a property read/write instead of `init`-only:

C#

```
public record Product
{
    public required string Name { get; init; }
    public decimal Price { get; set; }
}
```

`record class` vs. `record struct`

Because the `record` modifier preserves the underlying type's semantics, a `record class` and a `record struct` behave differently when you assign or compare references. Assigning a `record`

`class` copies the reference. Both variables point to the same object. Assigning a `record struct` copies the data, so changes to one variable don't affect the other:

C#

```
// Record class — assignment copies the reference
var p1 = new Person("Grace", "Hopper");
var p2 = p1; // p1 and p2 point to the same object:
Console.WriteLine(ReferenceEquals(p1, p2)); // True

// Record struct — assignment copies the data
var c1 = new Coordinate(47.6062, -122.3321);
var c2 = c1;
c2.Longitude = 0.0; // mutating c2 doesn't affect c1
Console.WriteLine(c1.Longitude); // -122.3321
Console.WriteLine(c2.Longitude); // 0
```

Record structs also provide compiler-generated value equality:

C#

```
var home = new Coordinate(47.6062, -122.3321);
var copy = home;

Console.WriteLine(home);           // Coordinate { Latitude = 47.6062, Longitude =
-122.3321 }
Console.WriteLine(home == copy);   // True — value equality
```

Choose `record class` when you need inheritance or when instances are large enough that copying would be expensive. Choose `record struct` for small, self-contained data where value-type copy semantics are appropriate. For more on value type semantics, see Structs.

Value equality

The `record` modifier gives both classes and structs compiler-generated, property-by-property equality. Here's how equality works across all four type kinds:

- **Plain class**: Uses *reference equality* by default. The `==` operator checks whether two variables point to the same object, not whether the data matches.
- **Plain struct**: Supports value equality through ValueType.Equals, but the default implementation uses reflection, which is slower and doesn't generate `==`/`!=` operators.
- **`record class`**: The compiler generates `Equals` and `GetHashCode` methods, and `==`/`!=` operators, that compare every property value. Two distinct objects with the same data are equal.

- `record struct`: Same compiler-generated equality as a `record class`, but without using reflection, which makes it faster than plain struct equality.

The following example demonstrates record class equality:

C#

```
// Person is a record type with three properties: FirstName, LastName, and
PhoneNumbers.
var phones = new string[] { "555-1234" };
var person1 = new Person("Grace", "Hopper", phones);
var person2 = new Person("Grace", "Hopper", phones);

Console.WriteLine(person1 == person2);              // True
Console.WriteLine(ReferenceEquals(person1, person2)); // False

person1.PhoneNumbers[0] = "555-9999";
Console.WriteLine(person2.PhoneNumbers[0]); // 555-9999 — same array
```

The two `Person` instances are different objects, but they're equal because all their property values match. Array properties compare by reference, not by contents. Mutating the shared array is visible through both records because the array itself isn't copied.

Nondestructive mutation with `with` expressions

Records are often immutable, so you can't change a property after creation. A `with` expression creates a copy with one or more properties changed, leaving the original record unchanged. This approach works for both `record class` and `record struct` types:

C#

```
var original = new Person("Grace", "Hopper");
var modified = original with { FirstName = "Margaret" };

Console.WriteLine(original); // Person { FirstName = Grace, LastName = Hopper }
Console.WriteLine(modified); // Person { FirstName = Margaret, LastName = Hopper }
Console.WriteLine(original == modified); // False

var copy = original with { };
Console.WriteLine(original == copy); // True
```

The same syntax works for `record struct` types:

C#

```
var shifted = home with { Longitude = -122.0 };
Console.WriteLine(shifted);        // Coordinate { Latitude = 47.6062, Longitude =
-122 }
Console.WriteLine(home == shifted); // False
```

A `with` expression copies the existing instance, then applies the specified property changes.

Positional records and deconstruction

Positional records generate a `Deconstruct` method that you use to extract property values into individual variables:

C#

```
var (first, last) = person;
Console.WriteLine($"{first} {last}");
// Grace Hopper
```

Deconstruction works with both `record class` and `record struct` types. You can use it in assignments, `foreach` loops, and pattern matching.

Record inheritance

A `record class` can inherit from another `record class`. A record can't inherit from a regular class, and a class can't inherit from a record:

C#

```
public record Student(string FirstName, string LastName, int GradeLevel)
    : Person(FirstName, LastName);
```

Value equality checks include the run-time type, so a `Person` and a `Student` with the same `FirstName` and `LastName` aren't considered equal. Record structs don't support inheritance because structs can't inherit from other types.

See also

- Type system overview
- Classes
- Structs
- Records (C# reference)

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/06/2026

Interfaces - define behavior for multiple types

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You'll encounter interfaces once you need to define shared behavior across unrelated types.
>
> **Experienced in another language?** C# interfaces are similar to interfaces in Java or protocols in Swift. Skim the explicit implementation section for C#-specific patterns.

An *interface* defines a contract: a group of related methods, properties, events, and indexers that a class or struct must implement. Interfaces let a single type implement multiple contracts, which is important because C# doesn't support multiple inheritance of classes. Structs can't inherit from other structs or classes, so interfaces are the only way to add shared behavior across struct types.

The following example declares an interface and a class that implements it:

C#

```
interface IEquatable<T>
{
    bool Equals(T obj);
}
```

C#

```
public class Car : IEquatable<Car>
{
    public string? Make { get; set; }
    public string? Model { get; set; }
    public string? Year { get; set; }

    public bool Equals(Car? car) =>
        car is not null &&
        (Make, Model, Year) == (car.Make, car.Model, car.Year);
}
```

Any class or struct that implements IEquatable<T> must provide an `Equals` method that matches the interface signature. You can count on any `IEquatable<T>` implementation to support equality comparison, regardless of the concrete type. That predictability is the core value of interfaces.

Declare an interface

Define an interface with the interface keyword. By convention, interface names begin with a capital `I`:

C#

```
interface ILogger
{
    void Log(string message);
    string Name { get; }
}
```

Interfaces can contain methods, properties, events, and indexers. An interface can't contain instance fields, instance constructors, or finalizers. Members are `public` by default. You can specify other accessibility modifiers when needed. For example, use `internal` for members that shouldn't be visible outside the assembly.

Implement an interface

A class or struct lists the interfaces it implements after a colon in its declaration. The class must provide an implementation for every member declared in the interface:

C#

```
public class ConsoleLogger : ILogger
{
    public string Name => "Console";

    public void Log(string message) =>
        Console.WriteLine($"[{Name}] {message}");
}

public class FileLogger : ILogger
{
    public string Name => "File";

    public void Log(string message)
    {
        // In a real app, write to a file
```

```
        Console.WriteLine($"[{Name}] Writing to file: {message}");
    }
}
```

A class can implement multiple interfaces, separated by commas. It must provide implementations for all members from every interface it lists.

Explicit implementation

Sometimes you need to implement an interface member without making it part of the class's public API. *Explicit implementation* qualifies the member with the interface name. The member is only accessible through a variable of the interface type:

C#

```
interface IMetric
{
    double GetDistance(); // Returns meters
}

interface IImperial
{
    double GetDistance(); // Returns feet
}

public class Runway(double meters) : IMetric, IImperial
{
    // Explicit implementation for IMetric
    double IMetric.GetDistance() => meters;

    // Explicit implementation for IImperial
    double IImperial.GetDistance() => meters * 3.28084;
}
```

Explicit implementation is useful when two interfaces declare members with the same name, or when you want to keep the class's public surface clean. For more detail, see Explicit Interface Implementation.

Interface inheritance

Interfaces can inherit from one or more other interfaces. A class that implements a derived interface must implement all members from the derived interface and all of its base interfaces:

C#

```
interface IDrawable
{
    void Draw();
}

interface IShape : IDrawable
{
    double Area { get; }
}

public class Circle(double radius) : IShape
{
    public double Area => Math.PI * radius * radius;

    public void Draw() =>
        Console.WriteLine($"Drawing circle with area {Area:F2}");
}
```

A class that implements `IShape` can be implicitly converted to `IDrawable`, because `IShape` inherits from it.

Interfaces vs. abstract classes

Both interfaces and abstract classes define contracts that derived types must fulfill.

- **Use an abstract class** when related types share state (fields), constructors, or non-public members. Abstract classes let you evolve a hierarchy by adding new members with default behavior without breaking existing derived types.
- **Use an interface** when a type needs to fulfill a contract that cuts across unrelated hierarchies, or when it needs to implement multiple contracts. Interfaces can't declare instance fields or constructors, so they're best suited for adding capabilities to types that already have a base class. For advanced scenarios, interfaces also support default member implementations.

A class can inherit from only one base class but can implement multiple interfaces. That distinction often makes interfaces the better choice for defining capabilities that cut across type hierarchies.

Working with internal interfaces

You can typically implement an internal interface with public members, as long as all types in the interface signature are publicly accessible. When an interface uses internal types in its

member signatures, you must use explicit implementation because the implementing member can't be public while exposing internal types:

C#

```
internal class InternalConfiguration
{
    public string Setting { get; set; } = "";
}

internal interface ILoggable
{
    void Log(string message);
}

internal interface IConfigurable
{
    void Configure(InternalConfiguration config);
}

public class ServiceImplementation : ILoggable, IConfigurable
{
    // Implicit implementation: ILoggable uses only public types in its signature
    public void Log(string message) =>
        Console.WriteLine($"Log: {message}");

    // Explicit implementation: IConfigurable uses internal types
    void IConfigurable.Configure(InternalConfiguration config) =>
        Console.WriteLine($"Configured with: {config.Setting}");
}
```

In the preceding example, `IConfigurable` uses an internal type `InternalConfiguration` in its method signature. `ServiceImplementation` uses explicit implementation for that member. In contrast, `ILoggable` uses only public types (`string`) in its signature and can be implemented implicitly.

Default interface members and static abstract members

Interfaces support two advanced features that go beyond basic contracts:

- **Default interface members** let an interface provide a method body. Implementing types inherit the default implementation and can optionally override it. For more information, see default interface methods.
- **Static abstract members** require implementing types to provide a static member, which is useful for defining operator contracts or factory patterns. For more information, see static

abstract members in interfaces.

Both features are covered in the article on interfaces in the language reference. Most everyday interface usage involves the declaring and implementing patterns described earlier in this article.

Interfaces summary

- An interface defines a contract of methods, properties, events, and indexers.
- A class or struct that implements an interface must provide implementations for all declared members (unless the interface provides a default implementation).
- You can't instantiate an interface directly.
- A class or struct can implement multiple interfaces. A class can inherit a base class and also implement one or more interfaces.
- Interface names conventionally start with `I`.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/13/2026

C# enumerations

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You'll encounter enums once you need to represent a fixed set of choices in your code.
>
> **Experienced in another language?** C# enums work similarly to enums in Java or C++, with additional support for bit flags and pattern matching. Skim the flags and switch expression sections for C#-specific patterns.

An *enumeration type* (or *enum*) defines a set of named constants backed by an integer value. Use enums when a value must be one of a fixed set of options, such as days of the week, HTTP status codes, log levels, or directions. Enums make your code more readable and less error-prone than raw integer constants because the compiler enforces the named values.

Declare an enum

Define an enum with the `enum` keyword followed by the type name and its members:

C#

```
enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

By default, the underlying type is `int`, and values start at `0` and increment by one. `Season.Spring` is `0`, `Season.Summer` is `1`, and so on.

Specify an underlying type and explicit values

You can choose a different integral type and assign explicit values to control the numeric representation:

C#

```
enum HttpStatus : ushort
{
    OK = 200,
    NotFound = 404,
    InternalServerError = 500
}
```

Use explicit values when the numbers have external meaning, such as HTTP status codes or protocol identifiers. The underlying type can be any integral type except `char`. Use `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `ulong`.

Use enums in switch expressions

Enums work naturally with `switch` expressions and pattern matching. The compiler warns you if you don't handle all members, which helps prevent bugs when you add a new value later:

C#

```
static string DescribeSeason(Season season) => season switch
{
    Season.Spring => "Flowers bloom and temperatures rise.",
    Season.Summer => "Long days and warm weather.",
    Season.Autumn => "Leaves change color and fall.",
    Season.Winter => "Short days and cold temperatures.",
    _ => throw new ArgumentOutOfRangeException(nameof(season))
};
```

C#

```
var today = Season.Autumn;
Console.WriteLine(DescribeSeason(today));
```

The discard pattern (`_`) handles any value not explicitly listed. Because an enum's underlying type is an integer, a variable can hold a value that doesn't correspond to any named member. For example, `(Season)99` is valid at runtime. The discard pattern ensures the switch expression handles those unexpected values safely. *Pattern matching* is a C# feature that tests a value against a shape or condition. In this example, each `case` checks whether the enum matches a specific member. Switch expressions are one of several pattern matching forms. For more information about pattern matching, see Pattern matching.

Bit flags

When an enum represents a combination of choices rather than a single choice, define each member as a power of two and apply the FlagsAttribute:

C#

```
[Flags]
enum FileAccess
{
    None = 0,
    Read = 1,
    Write = 2,
    Execute = 4,
    ReadWrite = Read | Write,
    All = Read | Write | Execute
}
```

Combine values by using the `|` operator and test for individual flags by using HasFlag:

C#

```
var permissions = FileAccess.Read | FileAccess.Write;

Console.WriteLine(permissions);                          // ReadWrite
Console.WriteLine(permissions.HasFlag(FileAccess.Read)); // True
Console.WriteLine(permissions.HasFlag(FileAccess.Execute)); // False
```

The `[Flags]` attribute also affects `ToString()`. It displays combined values as comma-separated names (like `Read, Write`) instead of a raw number. For more information, see System.FlagsAttribute.

Convert between enums and integers

Explicit cast expressions convert between an enum and its underlying integer type. An explicit cast uses the `(Type)value` syntax to tell the compiler you intend the conversion:

C#

```
var status = HttpStatus.NotFound;
ushort code = (ushort)status;
Console.WriteLine($"Status: {status} ({code})"); // Status: NotFound (404)

var fromCode = (HttpStatus)200;
Console.WriteLine(fromCode); // OK
```

Be aware that casting an integer to an enum doesn't validate whether the value matches a defined member. Use Enum.IsDefined to check validity when you accept numeric input from

external sources.

Parse strings and iterate values

The Enum base class provides methods for parsing strings and iterating over all defined values:

C#

```
// Parse a string to an enum value:
var parsed = Enum.Parse<Season>("Winter");
Console.WriteLine(parsed); // Winter

// Try to parse safely. It returns false only when the input can't be parsed. Call
Enum.IsDefined to validate named members:
if (Enum.TryParse<Season>("Monsoon", out var unknown))
{
    Console.WriteLine(unknown);
}
else
{
    Console.WriteLine("'Monsoon' is not a valid Season"); // 'Monsoon' is not a
valid Season
}

// Iterate over all values in an enum:
foreach (var season in Enum.GetValues<Season>())
{
    Console.WriteLine($"{season} = {(int)season}");
}
// Spring = 0
// Summer = 1
// Autumn = 2
// Winter = 3
```

Use Enum.TryParse<TEnum>(String, Boolean, TEnum) instead of Enum.Parse<TEnum>(String) when the input might be invalid. It returns `false` instead of throwing an exception.

See also

- Type system overview
- Enumeration types (C# reference)
- Pattern matching
- Enum design guidelines

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Last updated on 04/13/2026

Generic types and methods

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You'll encounter generics as soon as you use collections like `List<T>`.
>
> **Experienced in another language?** C# generics are similar to generics in Java or templates in C++, but with full runtime type information and no type erasure. Skim the collection expressions and covariance and contravariance sections for C#-specific patterns.

Generics let you write code that works with any type while keeping full type safety. Instead of writing separate classes or methods for `int`, `string`, and every other type you need, write one version with one or more *type parameters* (such as `T`, or `TKey` and `TValue`) and specify the actual types when you use it. The compiler checks types at compile time, so you don't need runtime casts or risk `InvalidCastException`.

You encounter generics constantly in everyday C#. Collections, async return types, delegates, and LINQ all rely on generic types:

C#

```
List<int> scores = [95, 87, 72, 91];
Dictionary<string, decimal> prices = new()
{
    ["Widget"] = 19.99m,
    ["Gadget"] = 29.99m
};
Task<string> greeting = Task.FromResult("Hello, generics!");
Func<int, bool> isPositive = n => n > 0;

Console.WriteLine($"First score: {scores[0]}");
Console.WriteLine($"Widget price: {prices["Widget"]:C}");
Console.WriteLine($"Greeting: {await greeting}");
Console.WriteLine($"Is 5 positive? {isPositive(5)}");
```

In each case, the type argument in angle brackets (`<int>`, `<string>`, `<Product>`) tells the generic type what kind of data it holds or operates on. The compiler enforces type safety. You can't accidentally add a `string` to a `List<int>`.

Consuming generic types

More often, you *consume* generic types from the .NET class library rather than creating your own. The following sections show the most common generic types you'll use.

Generic collections

The System.Collections.Generic namespace provides type-safe collection classes. Always use these collections instead of nongeneric collections like ArrayList:

C#

```
// A strongly typed list of strings
List<string> names = ["Alice", "Bob", "Carol"];
names.Add("Dave");
// names.Add(42); // Compile-time error: can't add an int to List<string>

// A dictionary mapping string keys to int values
var inventory = new Dictionary<string, int>
{
    ["Apples"] = 50,
    ["Oranges"] = 30
};
inventory["Bananas"] = 25;

// A set that prevents duplicates
HashSet<int> uniqueIds = [1, 2, 3, 1, 2];
Console.WriteLine($"Unique count: {uniqueIds.Count}"); // 3

// A FIFO queue
Queue<string> tasks = new();
tasks.Enqueue("Build");
tasks.Enqueue("Test");
Console.WriteLine($"Next task: {tasks.Dequeue()}"); // Build
```

Generic collections prevent type errors at runtime because the errors surface at compile time instead. These collections also avoid boxing for value types, which improves performance.

Generic methods

A generic method declares its own type parameter. The compiler often *infers* the type argument from the values you pass, so you don't need to specify it explicitly:

C#

```
static void Print<T>(T value) =>
    Console.WriteLine($"Value: {value}");

Print(42);        // Compiler infers T as int
```

```
Print("hello");   // Compiler infers T as string
Print(3.14);      // Compiler infers T as double
```

In the call `Print(42)`, the compiler infers `T` as `int` from the argument. You can write `Print<int>(42)` explicitly, but type inference keeps the code cleaner.

Collection expressions

Collection expressions (C# 12) provide a concise syntax for creating collections. Use square brackets instead of constructor calls or initializer syntax:

C#

```
// Create a list with a collection expression
List<string> fruits = ["Apple", "Banana", "Cherry"];

// Create an array
int[] numbers = [1, 2, 3, 4, 5];

// Works with any supported collection type
IReadOnlyList<double> temperatures = [72.0, 68.5, 75.3];

Console.WriteLine($"Fruits: {string.Join(", ", fruits)}");
Console.WriteLine($"Numbers: {string.Join(", ", numbers)}");
Console.WriteLine($"Temps: {string.Join(", ", temperatures)}");
```

The *spread operator* (`..`) inlines the elements of one collection into another, which is useful for combining sequences:

C#

```
List<int> first = [1, 2, 3];
List<int> second = [4, 5, 6];

// Spread both lists into a new combined list
List<int> combined = [.. first, .. second];
Console.WriteLine(string.Join(", ", combined));
// Output: 1, 2, 3, 4, 5, 6

// Add extra elements alongside spreads
List<int> withExtras = [0, .. first, 99, .. second];
Console.WriteLine(string.Join(", ", withExtras));
// Output: 0, 1, 2, 3, 99, 4, 5, 6
```

Collection expressions work with arrays, `List<T>`, `Span<T>`, `ImmutableArray<T>`, and any type that supports the collection builder pattern. For the complete syntax reference, see Collection expressions.

Dictionary initialization

You can initialize dictionaries concisely with indexer initializers. This syntax uses square brackets to set key-value pairs:

C#

```
Dictionary<string, int> scores = new()
{
    ["Alice"] = 95,
    ["Bob"] = 87,
    ["Carol"] = 92
};

foreach (var (name, score) in scores)
{
    Console.WriteLine($"{name}: {score}");
}
```

You can merge dictionaries by copying one and applying overrides:

C#

```
Dictionary<string, int> defaults = new()
{
    ["Timeout"] = 30,
    ["Retries"] = 3
};
Dictionary<string, int> overrides = new()
{
    ["Timeout"] = 60
};

// Merge defaults and overrides into a new dictionary
Dictionary<string, int> config = new(defaults);
foreach (var (key, value) in overrides)
{
    config[key] = value;
}

Console.WriteLine($"Timeout: {config["Timeout"]}");  // 60
Console.WriteLine($"Retries: {config["Retries"]}");   // 3
```

Type constraints

Constraints restrict which type arguments a generic type or method accepts. Constraints let you call methods or access properties on the type parameter that wouldn't be available on `object` alone:

C#

```
static T Max<T>(T a, T b) where T : IComparable<T> =>
    a.CompareTo(b) >= 0 ? a : b;

Console.WriteLine(Max(3, 7));          // 7
Console.WriteLine(Max("apple", "banana")); // banana

static T CreateDefault<T>() where T : new() => new T();

var list = CreateDefault<List<int>>(); // Creates an empty List<int>
Console.WriteLine($"Empty list count: {list.Count}"); // 0
```

The most common constraints are:

Expand table

Constraint	Meaning
`where T : class`	`T` must be a reference type
`where T : struct`	`T` must be a non-nullable value type
`where T : new()`	`T` must have a public parameterless constructor
`where T : BaseClass`	`T` must derive from `BaseClass`
`where T : IInterface`	`T` must implement `IInterface`

You can combine constraints: `where T : class, IComparable<T>, new()`. Less common constraints include `where T : System.Enum`, `where T : System.Delegate`, and `where T : unmanaged` for specialized scenarios. For the complete list, see Constraints on type parameters.

Covariance and contravariance

Covariance and *contravariance* describe how generic types behave with inheritance. They determine whether you can use a more derived or less derived type argument than originally specified:

C#

```
// Covariance: IEnumerable<Dog> can be used as IEnumerable<Animal>
// because IEnumerable<out T> is covariant
List<Dog> dogs = [new("Rex"), new("Buddy")];
IEnumerable<Animal> animals = dogs; // Allowed because Dog derives from Animal

foreach (var animal in animals)
{
```

```
    Console.WriteLine(animal.Name);
}

// Contravariance: Action<Animal> can be used as Action<Dog>
// because Action<in T> is contravariant
Action<Animal> printAnimal = a => Console.WriteLine($"Animal: {a.Name}");
Action<Dog> printDog = printAnimal; // Allowed because any Animal handler can
handle Dog

printDog(new Dog("Spot"));
```

- **Covariance** (`out T`): An `IEnumerable<Dog>` can be used where `IEnumerable<Animal>` is expected because `Dog` derives from `Animal`. The `out` keyword on the type parameter enables this. Covariant type parameters can only appear in output positions (return types).
- **Contravariance** (`in T`): An `Action<Animal>` can be used where `Action<Dog>` is expected because any action that handles `Animal` can also handle `Dog`. The `in` keyword enables this. Contravariant type parameters can only appear in input positions (parameters).

Many built-in interfaces and delegates are already variant: `IEnumerable<out T>`, `IReadOnlyList<out T>`, `Func<out TResult>`, and `Action<in T>`. You benefit from variance automatically when working with these types. For an in-depth treatment of designing variant interfaces and delegates, see Covariance and contravariance.

Create your own generic types

You can define your own generic classes, structs, interfaces, and methods. The following example shows a simple generic linked list for illustration. In practice, use List<T> or another built-in collection:

C#

```
public class GenericList<T>
{
    private class Node(T data)
    {
        public T Data { get; set; } = data;
        public Node? Next { get; set; }
    }

    private Node? head;

    public void AddHead(T data)
    {
        var node = new Node(data) { Next = head };
        head = node;
```

```
    }

    public IEnumerator<T> GetEnumerator()
    {
        var current = head;
        while (current is not null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }
}
```

C#

```
var list = new GenericList<int>();
for (var i = 0; i < 5; i++)
{
    list.AddHead(i);
}

foreach (var item in list)
{
    Console.Write($"{item} ");
}
Console.WriteLine();
// Output: 4 3 2 1 0
```

Generic types aren't limited to classes. You can define generic `interface`, `struct`, and `record` types. For more information about designing generic algorithms and complex constraint combinations, see Generics in .NET.

See also

- Generics in .NET
- Collection expressions
- Constraints on type parameters
- Covariance and contravariance
- System.Collections.Generic

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/13/2026

Tuples and deconstruction

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials. You'll encounter tuples when you need to return multiple values from a method or group values without defining a named type.
>
> **Experienced in another language?** C# tuples are value types similar to tuples in Python or Swift, but with optional named elements and full deconstruction support. Skim the deconstruction and equality sections for C#-specific patterns.

A *tuple* groups multiple values into a single, lightweight structure without requiring you to define a named type. Tuples are value types that you can declare inline, return from methods, and deconstruct into individual variables. Use tuples when you need a quick, temporary grouping of related values. For example, when you return multiple results from a method or store a coordinate pair.

The following example creates a tuple with named elements and accesses each element by name:

C#

```
var location = (Latitude: 47.6062, Longitude: -122.3321);
Console.WriteLine($"Location: {location.Latitude}, {location.Longitude}");
// Output: Location: 47.6062, -122.3321
```

Tuples work well for short-lived groupings where defining a class, struct, or record would add unnecessary ceremony. For long-lived domain concepts or types with behavior, prefer records, classes, or structs. For a comparison of when to use each, see Choose which kind of type.

Declare and initialize tuples

Declare a tuple by listing the element types in parentheses. You can optionally name each element to make the code more readable:

C#

```
// Tuple with named elements
(string Name, int Age) person = ("Alice", 30);
Console.WriteLine($"{person.Name} is {person.Age} years old");

// Tuple with default element names (Item1, Item2)
(string, int) unnamed = ("Bob", 25);
Console.WriteLine($"{unnamed.Item1} is {unnamed.Item2} years old");

// Tuple declared with var and inline names
var city = (Name: "Seattle", Population: 749_256);
Console.WriteLine($"{city.Name}: population {city.Population}");
```

When you don't provide names, elements use default names `Item1`, `Item2`, and so on. Named elements make your code self-documenting without requiring a separate type definition.

Inferred element names

The compiler infers element names from the variable names or property names you use to initialize the tuple. This feature avoids redundancy when the names match:

C#

```
var name = "Carol";
var age = 28;

// The compiler infers element names from the variable names
var person = (name, age);
Console.WriteLine($"{person.name} is {person.age}");
// Output: Carol is 28
```

Inferred names keep your code concise. If you need a different element name, specify it explicitly.

Return multiple values from a method

One of the most common uses for tuples is returning multiple values from a method. Instead of defining a class or using `out` parameters, return a tuple with named elements:

C#

```
static (double Minimum, double Maximum, double Average) ComputeStats(List<double>
values)
{
    var min = values.Min();
    var max = values.Max();
    var avg = values.Average();
```

```
    return (min, max, avg);
}
```

Named tuple elements make the return values readable at both the call site and the method signature. The caller can access each value by name without needing to remember positional order.

Deconstruct tuples

Deconstruction unpacks a tuple's elements into separate variables in a single statement. You can deconstruct tuples in several ways:

C#

```
var point = (X: 3, Y: 7);

// Deconstruct with var (infer all types)
var (x, y) = point;
Console.WriteLine($"x={x}, y={y}");

// Deconstruct with explicit types
(int px, int py) = point;
Console.WriteLine($"px={px}, py={py}");

// Deconstruct into existing variables
int a, b;
(a, b) = point;
Console.WriteLine($"a={a}, b={b}");

// Deconstruct a method return value directly
List<double> data = [10.0, 20.0, 30.0];
var (min, max, avg) = ComputeStats(data);
Console.WriteLine($"Min: {min}, Max: {max}, Avg: {avg}");
```

Deconstruction is especially useful when you receive a tuple from a method call and immediately need to work with its individual values.

You can deconstruct tuples directly in `foreach` loops, which makes iterating over collections of grouped values concise:

C#

```
List<(string Name, int Score)> results =
[
    ("Alice", 92),
    ("Bob", 87),
    ("Carol", 95)
```

```
];

foreach (var (name, score) in results)
{
    Console.WriteLine($"{name}: {score}");
}
```

When you don't need every element, use a *discard* (`_`) in place of each value you want to ignore. Use a separate `_` for each discarded position:

C#

```
List<double> values = [5.0, 10.0, 15.0];
var (_, max, _) = ComputeStats(values);
Console.WriteLine($"Only need the max: {max}");
// Output: Only need the max: 15
```

For more about using discards across different contexts, see Discards.

Tuple equality

You can compare tuples using `==` and `!=`. These operators compare each element in order, so two tuples are equal when all their corresponding elements are equal:

C#

```
var order1 = (Product: "Widget", Quantity: 5);
var order2 = (Product: "Widget", Quantity: 5);
var order3 = (Product: "Gadget", Quantity: 3);

Console.WriteLine(order1 == order2); // True
Console.WriteLine(order1 == order3); // False

// Element names don't affect equality—only values matter
var named = (X: 1, Y: 2);
var different = (A: 1, B: 2);
Console.WriteLine(named == different); // True
```

Tuple equality uses the `==` operator defined on each element type, which means the comparison works correctly for strings, numbers, and other types that define `==`. Element names don't affect equality—only values and positions matter.

Nondestructive mutation with `with`

The `with` expression creates a copy of a tuple with one or more elements changed, leaving the original unchanged:

C#

```
var original = (Name: "Widget", Price: 19.99m, InStock: true);
var discounted = original with { Price = 14.99m };

Console.WriteLine($"Original: {original.Name} at {original.Price:C}");
Console.WriteLine($"Discounted: {discounted.Name} at {discounted.Price:C}");
// Output:
// Original: Widget at $19.99
// Discounted: Widget at $14.99
```

This pattern is useful when you want a variation of an existing tuple without modifying the original. The `with` expression works the same way on tuples as it does on records.

Tuples in dictionaries and lookups

Tuples make convenient dictionary values when you need to associate a key with multiple pieces of data:

C#

```
var sizeChart = new Dictionary<string, (int Min, int Max)>
{
    ["Small"] = (0, 50),
    ["Medium"] = (51, 100),
    ["Large"] = (101, 200)
};

if (sizeChart.TryGetValue("Medium", out var range))
{
    Console.WriteLine($"Medium: {range.Min}–{range.Max}");
}
// Output: Medium: 51–100
```

Tuples also work as dictionary *keys*, giving you a composite key without defining a custom type. Because tuples implement structural equality, lookups match on the combined values of all elements:

C#

```
var grid = new Dictionary<(int Row, int Column), string>
{
    [(0, 0)] = "Origin",
    [(1, 3)] = "Sensor A",
```

```
    [(2, 5)] = "Sensor B"
};

var target = (Row: 1, Column: 3);
if (grid.TryGetValue(target, out var label))
{
    Console.WriteLine($"({target.Row}, {target.Column}): {label}");
}
// Output: (1, 3): Sensor A
```

This pattern avoids needing a separate class for simple multi-key lookups or key-to-multiple-value mappings.

Tuples vs. anonymous types

Tuples are the preferred choice when you need a lightweight unnamed data structure. Anonymous types remain available for expression tree scenarios and for code that requires reference types, but tuples offer better performance, deconstruction support, and more flexible syntax. For more about anonymous types, see Choosing between anonymous and tuple types.

See also

- Tuple types (C# reference) for complete syntax details
- Deconstructing tuples and other types for user-defined `Deconstruct` methods
- Discards
- Records

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/13/2026

Type conversions, casting, and boxing

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You practice basic types there before you make conversion choices.
>
> **Experienced in another language?** C# conversions work like most statically typed languages: widening conversions are implicit, narrowing conversions need explicit casts, and parsing text should favor `TryParse` in user-facing code.

When you write C# code, you often convert values from one type to another. For example, you might convert from `int` to `long`, read text and convert it to a number, or cast a base type to a derived type.

Understanding key terms:

- A *conversion* is the process of changing a value from one type to another.
- A *cast* is the explicit syntax for conversion, written with parentheses like `(int)value`.
- An *implicit conversion* is a conversion that happens automatically when the compiler can guarantee it's safe.
- An *explicit cast* is a conversion you write in code, indicating the conversion might lose information or fail.

Match the conversion style to the situation:

- Implicit conversions happen automatically when the compiler can guarantee safety—no syntax required.
- Explicit casts are required when data might be lost or the conversion might fail.
- Pattern matching or `as` applies when you need a safe reference-type conversion that might not succeed.
- Parsing APIs apply when your source value is text.

Implicit and explicit numeric conversions

An *implicit conversion* always succeeds. An *explicit conversion* might fail or lose information.

C#

```
int itemCount = 42;
long widened = itemCount; // Implicit conversion.

double average = 19.75;
int truncated = (int)average; // Explicit cast.

Console.WriteLine($"widened: {widened}, truncated: {truncated}");
```

An explicit cast tells readers that the conversion might lose information. In the sample, the `double` value is truncated when converted to `int`.

For full conversion tables, see Built-in numeric conversions.

Convert references

Classes are reference types. Casts on them don't copy data. They change how you view the same object.

Some reference conversions are implicit. The compiler guarantees they're safe. Three situations always produce an implicit reference conversion: assigning a derived class instance to a base class variable (the derived type is a subtype of the base type), assigning a reference type instance to a variable of an interface the type implements, and assigning any reference type to an `object` variable.

C#

```
Mammal otter = new() { Name = "River otter" };
Animal animal = otter;           // Derived to base class: always safe.
INamed named = otter;            // Reference type to implemented interface: always
safe.
object obj = otter;              // Any reference type to object: always safe.

Console.WriteLine(named.Name);
```

Going the other direction, from a base type back to a derived type, requires an explicit check, because the object might not actually be the derived type you expect. Prefer pattern matching so the test and assignment happen together.

C#

```
Animal knownAnimal = new Mammal { Name = "River otter" };

if (knownAnimal is Mammal mammal)
{
    Console.WriteLine($"Pattern match succeeded: {mammal.Name}");
```

```
}

Animal unknownAnimal = new Reptile();
Console.WriteLine($"Can treat as mammal: {unknownAnimal is Mammal}");
```

Pattern matching keeps the successful cast variable in the smallest practical scope, which improves readability.

If you need a nullable result instead of a conditional branch, use `as`:

C#

```
object boxedMammal = new Mammal { Name = "Sea lion" };
Mammal? maybeMammal = boxedMammal as Mammal;
Console.WriteLine(maybeMammal is null ? "Not a mammal" : maybeMammal.Name);

object boxedReptile = new Reptile();
Mammal? noMammal = boxedReptile as Mammal;
Console.WriteLine(noMammal is null ? "Safe null result" : noMammal.Name);
```

Use `as` only with reference types and nullable value types. It returns `null` when conversion fails.

Understand boxing and unboxing

Boxing converts a struct or other value type to `object` or to an implemented interface type. Unboxing extracts the value type from that object reference.

C#

```
int temperature = 72;
object boxedTemperature = temperature; // Boxing.
int unboxedTemperature = (int)boxedTemperature; // Unboxing.

Packet packet = new(7);
ILabelled labelledPacket = packet; // Boxing through an interface reference.

Console.WriteLine($"Unboxed: {unboxedTemperature}, Label: {labelledPacket.Label}");
```

Boxing allocates memory on the managed heap, and unboxing requires a type check. In hot paths, avoid unnecessary boxing because it adds allocations and extra work.

Parse text by using Parse and TryParse

When you convert user input or file content, start with `TryParse`. It avoids exceptions for expected invalid input and makes failure handling explicit. All parsing APIs create a new object or

value type instance from the source string; they don't modify the source.

C#

```
string textValue = "512";
int parsed = int.Parse(textValue);

string userInput = "12x";
bool parsedSuccessfully = int.TryParse(userInput, out int safeValue);

Console.WriteLine($"parsed: {parsed}");
Console.WriteLine(parsedSuccessfully ? $"safe value: {safeValue}" : "Input is not a
valid number.");
```

Use `Parse` when input is guaranteed to be valid, such as controlled test data. Use `TryParse` for user input, network payloads, and file data.

Core conversion APIs

Use these APIs most often in everyday code:

- Int32.Parse(String)
- Int32.TryParse
- Double.TryParse
- Decimal.TryParse
- System.IConvertible

For advanced conversion behavior and all overloads, review the API reference for the specific target type.

See also

- Type system overview
- Built-in types and literals
- Pattern matching
- How to safely cast by using pattern matching and the is and as operators

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/30/2026

Lambda expressions, delegates, and events

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. Build core type and method skills there before you use lambda expressions.

Sometimes you want to pass a small piece of behavior, a function, directly to another method. For example, you might want to filter a list, but the filtering condition changes depending on the situation. Rather than writing a separate named method for every possible condition, you pass the condition itself as an argument.

Lambda expressions are the C# feature that makes this possible. A lambda expression is a compact, inline function that you write without giving it a name. You use the arrow operator `=>` to separate the parameter list from the body:

C#

```
x => x * 2
```

Reading left to right: `x` is the input parameter, `=>` means "goes to," and `x * 2` is the body. It computes the value returned. When there are no parameters or more than one, wrap them in parentheses: `() => 42` or `(left, right) => left + right`.

Delegates support lambda expressions

To use a lambda expression, the C# compiler needs to know two things: the types of the parameters and the return type. That description, the parameter types plus return type, is called a *delegate type*.

A delegate type is a type that represents a method signature. A variable of a delegate type can hold any matching method such as a lambda expression or a named method, as long as its parameter types and return type match.

You declare a delegate type with the `delegate` keyword:

C#

```
delegate int Transform(int value);
```

This declaration says: "`Transform` is a delegate type for methods that accept one `int` and return an `int`." You can then assign a lambda expression or a named method to a variable of that type:

C#

```
Transform doubler = x => x * 2;    // assign a lambda expression
Transform squarer = Square;         // assign a named method

Console.WriteLine(doubler(5));      // 10
Console.WriteLine(squarer(5));      // 25

static int Square(int value) => value * value;
```

Both `doubler` and `squarer` hold a value of type `Transform`. You call them exactly like regular methods. The compiler verifies that whatever you assign matches the declared signature.

Built-in delegate types: `Func` and `Action`

Declaring a custom delegate type for every situation can be repetitive. .NET provides two families of generic delegate types, `Func` and `Action`, that cover most scenarios, so you rarely need to use the `delegate` keyword yourself.

Both families come in versions with zero to sixteen input type parameters, so they scale to any number of inputs. The key difference between the two families is:

- System.Func<T,TResult> (and `Func<T1, T2, TResult>`, and so on) represents a method that **returns a value**. The last type parameter is always the return type; all earlier ones are input types.
- System.Action<T> (and `Action<T1, T2>`, and so on) represents a method that **returns nothing** (`void`). All type parameters are input types. System.Action with no type parameters represents a method with no inputs and no return value.

For example, `Func<int, int, int>` describes a method with two `int` inputs and one `int` result. `Action<string>` describes a method with one `string` input and no return value.

C#

```
Func<int, int, int> add = (left, right) => left + right;
Action<string> report = message => Console.WriteLine($"Report: {message}");
```

```
int total = add(5, 9);
report($"5 + 9 = {total}");
```

Use descriptive parameter names in lambdas so readers can understand intent without scanning the full method body.

Pass a lambda expression to a method

When a method declares a `Func` or `Action` parameter, callers pass a lambda expression that matches that delegate type. The compiler checks that the lambda's parameter types and return type match the declared delegate type. If they don't match, the code doesn't compile.

C#

```
int[] numbers = [1, 2, 3, 4, 5, 6];
int[] evenNumbers = Filter(numbers, value => value % 2 == 0).ToArray();

Console.WriteLine(string.Join(", ", evenNumbers));
```

The `Filter` method declares a `Func<int, bool>` parameter named `predicate`. The `Func<int, bool>` type tells callers the expected shape: one `int` input, one `bool` result. The caller passes `value => value % 2 == 0` as that argument. This pattern appears throughout LINQ and many .NET APIs.

Keep lambda expressions self-contained

A lambda expression can reference variables from the surrounding code. *Capturing* means the lambda holds a reference to a variable declared outside its own body. The combination of the lambda and the variables it captures is called a *closure*.

When you don't need to capture anything, add the `static` modifier to the lambda. A static lambda can only use its own parameters and values declared inside its body. It can't capture local variables or instance state from the enclosing scope.

C#

```
Func<int, bool> isEven = static value => value % 2 == 0;

Console.WriteLine(isEven(14));
Console.WriteLine(isEven(15));
```

Static lambdas make intent clear and prevent accidental captures.

Use discard parameters when inputs are irrelevant

Sometimes a delegate signature includes parameters you don't need. Use the discard `_` to signal that choice explicitly.

Common examples include event handlers where you don't use `sender` or `EventArgs`, callbacks where you only need some of several inputs, and LINQ overloads that provide an index you don't use.

C#

```
Action<int, int, string> statusUpdate = (_, _, message) => Console.WriteLine(message);
statusUpdate(200, 42, "Operation completed");
```

Discards improve readability because they show which parameters matter.

Events provide optional notifications

An *event* is a mechanism that one object (the *publisher*) uses to notify other objects (the *subscribers*) when something happens. The publisher doesn't need to know who is listening or how many subscribers there are. Subscribers choose to opt in.

Events are built on delegates. An event is a delegate field with extra restrictions enforced by the `event` keyword: outside code can only subscribe (`+=`) or unsubscribe (`-=`) from the event; only the class that declares the event can invoke (raise) it.

The .NET convention for event delegate types is System.EventHandler<TEventArgs>, where `T` is the type of data included with the notification. Its signature always has two parameters: the `sender` (the object that raised the event) and the event data of type `T`.

C#

```
MessagePublisher publisher = new();
publisher.MessagePublished += (_, message) => Console.WriteLine($"Received:
{message}");

publisher.Publish("Records updated");
```

Walking through the code:

- `MessagePublisher` declares `event EventHandler<string>? MessagePublished`. The `event` keyword means callers can only subscribe or unsubscribe—they can't invoke it directly.

- `publisher.MessagePublished += (_, message) => ...` subscribes with a lambda expression. The `_` discards the `sender` parameter because this handler doesn't need it.
- `publisher.Publish("Records updated")` raises the event and runs every subscribed handler.

Subscribing is optional. The `?.Invoke(...)` in the `Publish` method means the event raises only when at least one subscriber is attached. The publisher raises the event without knowing or caring whether anyone is listening.

See also

- Type system overview
- Methods
- Pattern matching
- Events (C# programming guide)

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/30/2026

Anonymous types (C# Programming Guide)

Anonymous types provide a convenient way to encapsulate a set of read-only properties into a single object without defining a named type first. The compiler generates a type name at compile time that you can't access in your source code. The compiler infers the type of each property.

Create anonymous types by using the new operator together with an object initializer. The following example shows an anonymous type that's initialized with two properties, `Name` and `Age`:

C#

```
var person = new { Name = "Alice", Age = 30 };
Console.WriteLine($"{person.Name} is {person.Age} years old.");
// Output:
// Alice is 30 years old.
```

Inferred property names

You can specify property names explicitly by using the `Name = value` syntax. When you initialize an anonymous type with a variable or member access expression, the compiler infers the property name from that expression:

C#

```
string productName = "Laptop";
decimal price = 999.99m;
var product = new { productName, price };
Console.WriteLine($"{product.productName}: {product.price:C}");
// Output:
// Laptop: $999.99
```

In the preceding example, the compiler infers `productName` and `price` as the property names from the variable names used in the initializer.

Declare anonymous types with var

Because the compiler generates the type name and you can't access it in source code, you must use var to declare the local variable. You can't specify the type name explicitly:

C#

```
// You must use var — you can't write a named type here.
var person = new { Name = "Alice", Age = 30 };
```

Use anonymous types in LINQ queries

Anonymous types appear most often in the select clause of a query expression, where they project a subset of properties from each source element:

C#

```
var words = new[] { "apple", "blueberry", "cherry" };

var results = words.Select(w => new { Word = w, Length = w.Length });

foreach (var item in results)
{
    Console.WriteLine($"{item.Word} has {item.Length} letters.");
}
// Output:
// apple has 5 letters.
// blueberry has 9 letters.
// cherry has 6 letters.
```

Equality

Two anonymous type instances that have the same property names and types in the same order share the same compiler-generated type. The compiler overrides Equals and GetHashCode so that equality compares property values rather than reference identity:

C#

```
var a = new { Name = "Alice", Age = 30 };
var b = new { Name = "Alice", Age = 30 };
var c = new { Name = "Bob", Age = 25 };

Console.WriteLine(a.Equals(b));  // True
Console.WriteLine(a.Equals(c));  // False
```

Nested anonymous types

Anonymous types can contain other anonymous types as property values:

C#

```
var order = new
{
    OrderId = 1,
    Customer = new { Name = "Alice", City = "Seattle" },
    Total = 150.00m
};
Console.WriteLine($"Order {order.OrderId} for {order.Customer.Name} in
{order.Customer.City}");
// Output:
// Order 1 for Alice in Seattle
```

Characteristics

Anonymous types have the following characteristics:

- The compiler generates them as `internal sealed class` types that derive from Object.
- All properties are `public` and read-only.
- Anonymous types support with expressions for nondestructive mutation.
- The compiler generates value-based Equals, GetHashCode, and ToString overrides.
- Anonymous types support expression trees, while tuples don't.

Limitations

Anonymous types have several limitations:

- You can't use them as method return types, method parameters, or field types because you can't name the type.
- They're scoped to the method where you declare them.
- You can't add methods, events, or custom operators.
- Properties are always read-only; anonymous types don't support mutable properties.

When to use tuples instead

For most new code, consider using tuples instead of anonymous types. As value types, tuples provide better performance. They also provide deconstruction support and more flexible syntax. Anonymous types remain the better choice when you need expression tree support or reference-type semantics. For a detailed comparison, see Choosing between anonymous and tuple types.

See also

- Object and collection initializers
- Implicitly typed local variables
- Tuples and deconstruction
- LINQ in C#
- Choosing between anonymous and tuple types

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/13/2026

Overview of object oriented techniques in C#

Article • 04/19/2025

In C#, the definition of a type—a class, struct, or record—is like a blueprint that specifies what the type can do. An object is basically a block of memory allocated and configured according to the blueprint. This article provides an overview of these blueprints and their features. The next article in this series introduces objects.

Encapsulation

Encapsulation is sometimes referred to as the first pillar or principle of object-oriented programming. A class or struct can specify how accessible each of its members is to code outside of the class or struct. Member not intended for consumers outside of the class or assembly are hidden to limit the potential for coding errors or malicious exploits. For more information, see the Object-oriented programming tutorial.

Members

The *members* of a type include all methods, fields, constants, properties, and events. In C#, there are no global variables or methods as there are in some other languages. Even a program's entry point, the `Main` method, must be declared within a class or struct (implicitly when you use top-level statements).

The following list includes all the various kinds of members that can be declared in a class, struct, or record.

- Fields
- Constants
- Properties
- Methods
- Constructors
- Events
- Finalizers
- Indexers
- Operators
- Nested Types

For more information, see Members.

Accessibility

Some methods and properties are meant to be called or accessed from code outside a class or struct, known as *client code*. Other methods and properties might be only for use in the class or struct itself. It's important to limit the accessibility of your code so that only the intended client code can reach it. You specify how accessible your types and their members are to client code by using the following access modifiers:

- public
- protected
- internal
- protected internal
- private
- private protected.

The default accessibility is `private`.

Inheritance

Classes (but not structs) support the concept of inheritance. A class that derives from another class, called the *base class*, automatically contains all the public, protected, and internal members of the base class except its constructors and finalizers.

Classes can be declared as abstract, which means that one or more of their methods have no implementation. Although abstract classes can't be instantiated directly, they can serve as base classes for other classes that provide the missing implementation. Classes can also be declared as sealed to prevent other classes from inheriting from them.

For more information, see Inheritance and Polymorphism.

Interfaces

Classes, structs, and records can implement multiple interfaces. To implement from an interface means that the type implements all the methods defined in the interface. For more information, see Interfaces.

Generic Types

Classes, structs, and records can be defined with one or more type parameters. Client code supplies the type when it creates an instance of the type. For example, the List<T> class in the System.Collections.Generic namespace is defined with one type parameter. Client code creates

an instance of a `List<string>` or `List<int>` to specify the type that the list holds. For more information, see Generics.

Static Types

Classes (but not structs or records) can be declared as `static`. A static class can contain only static members and can't be instantiated with the `new` keyword. One copy of the class is loaded into memory when the program loads, and its members are accessed through the class name. Classes, structs, and records can contain static members. For more information, see Static classes and static class members.

Nested Types

A class, struct, or record can be nested within another class, struct, or record. For more information, see Nested Types.

Partial Types

You can define part of a class, struct, or method in one code file and another part in a separate code file. For more information, see Partial Classes and Methods.

Object Initializers

You can instantiate and initialize class or struct objects, and collections of objects, by assigning values to its properties. For more information, see How to initialize objects by using an object initializer.

Anonymous Types

In situations where it isn't convenient or necessary to create a named class you use anonymous types. Named data members define anonymous types. For more information, see Anonymous types.

Extension Members

You can "extend" a class without creating a derived class by creating a separate type. That type contains methods that can be called as if they belonged to the original type. For more information, see Extension methods.

Implicitly Typed Local Variables

Within a class or struct method, you can use implicit typing to instruct the compiler to determine a variable's type at compile time. For more information, see var (C# reference).

Records

You can add the `record` modifier to a class or a struct. Records are types with built-in behavior for value-based equality. A record (either `record class` or `record struct`) provides the following features:

- Concise syntax for creating a reference type with immutable properties.
- Value equality. Two variables of a record type are equal if they have the same type, and if, for every field, the values in both records are equal. Classes use reference equality: two variables of a class type are equal if they refer to the same object.
- Concise syntax for nondestructive mutation. A `with` expression lets you create a new record instance that is a copy of an existing instance but with specified property values changed.
- Built-in formatting for display. The `ToString` method prints the record type name and the names and values of public properties.
- Support for inheritance hierarchies in record classes. Record classes support inheritance. Record structs don't support inheritance.

For more information, see Records.

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Objects - create instances of types

10/17/2025

A class or struct definition is like a blueprint that specifies what the type can do. An object is basically a block of memory that is allocated and configured according to the blueprint. A program might create many objects of the same class. Objects are also called instances, and they can be stored in either a named variable or in an array or collection. Client code is the code that uses these variables to call the methods and access the public properties of the object. In an object-oriented language such as C#, a typical program consists of multiple objects interacting dynamically.

> **Note**
>
> Static types behave differently than what is described here. For more information, see Static Classes and Static Class Members.

Struct Instances vs. Class Instances

Because classes are reference types, a variable of a class object holds a reference to the address of the object on the managed heap. If a second variable of the same type is assigned to the first variable, then both variables refer to the object at that address. This point is discussed in more detail later in this article.

Instances of classes are created by using the new operator. In the following example, `Person` is the type and `person1` and `person2` are instances, or objects, of that type.

C#

```
using System;

public class Person(string name, int age)
{
    public string Name { get; set; } = name;
    public int Age { get; set; } = age;
    // Other properties, methods, events...
}

class Program
{
    static void Main()
    {
        Person person1 = new("Leopold", 6);
        Console.WriteLine($"person1 Name = {person1.Name} Age = {person1.Age}");
```

```
        // Declare new person, assign person1 to it.
        Person person2 = person1;

        // Change the name of person2, and person1 also changes.
        person2.Name = "Molly";
        person2.Age = 16;

        Console.WriteLine($"person2 Name = {person2.Name} Age = {person2.Age}");
        Console.WriteLine($"person1 Name = {person1.Name} Age = {person1.Age}");

        /*
            Output:
            person1 Name = Leopold Age = 6
            person2 Name = Molly Age = 16
            person1 Name = Molly Age = 16
        */
    }
}
```

Because structs are value types, a variable of a struct object holds a copy of the entire object. Instances of structs can also be created by using the `new` operator, but this isn't required, as shown in the following example:

C#

```
using System;

namespace Example
{
    public struct Person
    {
        public string Name;
        public int Age;
        public Person(string name, int age)
        {
            Name = name;
            Age = age;
        }
    }

    public class Application
    {
        static void Main()
        {
            // Create  struct instance and initialize by using "new".
            // Memory is allocated on thread stack.
            Person p1 = new("Alex", 9);
            Console.WriteLine($"p1 Name = {p1.Name} Age = {p1.Age}");

            // Create  new struct object. Note that  struct can be initialized
            // without using "new".
            Person p2 = p1;
```

```
            // Assign values to p2 members.
            p2.Name = "Spencer";
            p2.Age = 7;
            Console.WriteLine($"p2 Name = {p2.Name} Age = {p2.Age}");

            // p1 values remain unchanged because p2 is  copy.
            Console.WriteLine($"p1 Name = {p1.Name} Age = {p1.Age}");
        }
    }
    /*
        Output:
        p1 Name = Alex Age = 9
        p2 Name = Spencer Age = 7
        p1 Name = Alex Age = 9
    */
}
```

The memory for both `p1` and `p2` is allocated on the thread stack. That memory is reclaimed along with the type or method in which it's declared. This is one reason why structs are copied on assignment. By contrast, the memory that is allocated for a class instance is automatically reclaimed (garbage collected) by the common language runtime when all references to the object are out of scope. It isn't possible to deterministically destroy a class object like you can in C++. For more information about garbage collection in .NET, see Garbage Collection.

> **Note**
>
> The allocation and deallocation of memory on the managed heap is highly optimized in the common language runtime. In most cases, there's no significant difference in the performance cost of allocating a class instance on the heap versus allocating a struct instance on the stack.

Object Identity vs. Value Equality

When you compare two objects for equality, you must first distinguish whether you want to know whether the two variables represent the same object in memory, or whether the values of one or more of their fields are equivalent. If you're intending to compare values, you must consider whether the objects are instances of value types (structs) or reference types (classes, delegates, arrays).

- To determine whether two class instances refer to the same location in memory (which means that they have the same *identity*), use the static Object.ReferenceEquals method. (System.Object is the implicit base class for all value types and reference types, including user-defined structs and classes.)

- The ValueType.Equals method, by default, determines whether the instance fields in two struct instances have the same values. Because all structs implicitly inherit from System.ValueType, you call the method directly on your object as shown in the following example:

  ```
  C#

  // Person is defined in the previous example.

  //public struct Person(string name, int age)
  //{
  //    public string Name { get; set; } = name;
  //    public int Age { get; set; } = age;
  //}

  Person p1 = new("Wallace", 75);
  Person p2 = new("", 42);
  p2.Name = "Wallace";
  p2.Age = 75;

  if (p2.Equals(p1))
      Console.WriteLine("p2 and p1 have the same values.");

  // Output: p2 and p1 have the same values.
  ```

  The default System.ValueType implementation of `Equals` uses boxing and reflection in some cases. For information about how to provide an efficient equality algorithm that's specific to your type, see How to define value equality for a type. Records are reference types that use value semantics for equality.

- To determine whether the values of the fields in two class instances are equal, you might be able to use the Equals method or the == operator. However, only use them if the class has overridden or overloaded them to provide a custom definition of what "equality" means for objects of that type. The class might also implement the IEquatable<T> interface or the IEqualityComparer<T> interface. Both interfaces provide methods that can be used to test value equality. When designing your own classes that override `Equals`, make sure to follow the guidelines stated in How to define value equality for a type and Object.Equals(Object).

Related Sections

For more information:

- Classes
- Constructors

- Finalizers
- Events
- object
- Inheritance
- class
- Structure types
- new Operator
- Common Type System

Inheritance - derive types to create more specialized behavior

Article • 02/16/2022

Inheritance, together with encapsulation and polymorphism, is one of the three primary characteristics of object-oriented programming. Inheritance enables you to create new classes that reuse, extend, and modify the behavior defined in other classes. The class whose members are inherited is called the *base class*, and the class that inherits those members is called the *derived class*. A derived class can have only one direct base class. However, inheritance is transitive. If `ClassC` is derived from `ClassB`, and `ClassB` is derived from `ClassA`, `ClassC` inherits the members declared in `ClassB` and `ClassA`.

> **Note**
>
> Structs do not support inheritance, but they can implement interfaces.

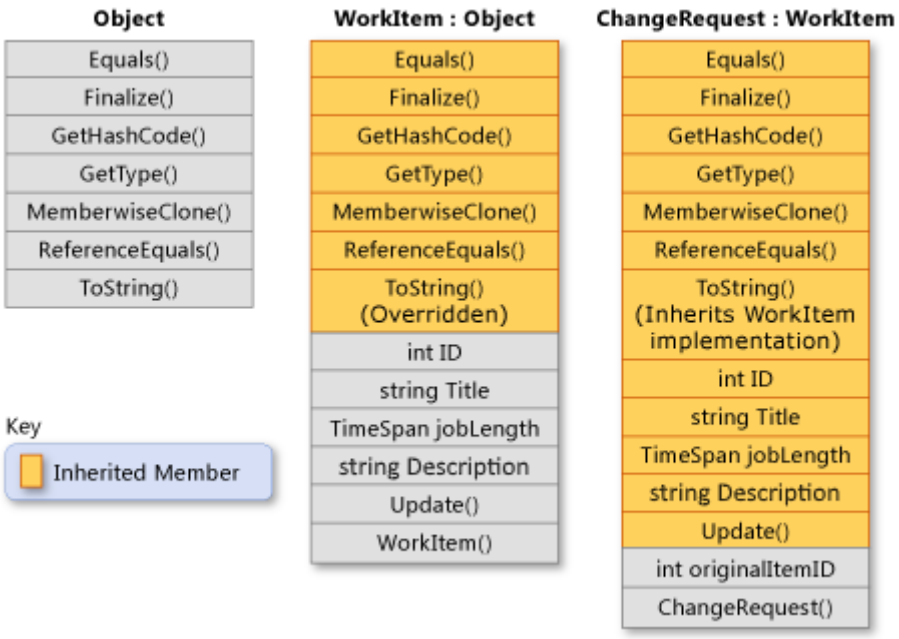
Conceptually, a derived class is a specialization of the base class. For example, if you have a base class `Animal`, you might have one derived class that is named `Mammal` and another derived class that is named `Reptile`. A `Mammal` is an `Animal`, and a `Reptile` is an `Animal`, but each derived class represents different specializations of the base class.

Interface declarations may define a default implementation for its members. These implementations are inherited by derived interfaces, and by classes that implement those interfaces. For more information on default interface methods, see the article on interfaces.

When you define a class to derive from another class, the derived class implicitly gains all the members of the base class, except for its constructors and finalizers. The derived class reuses the code in the base class without having to reimplement it. You can add more members in the derived class. The derived class extends the functionality of the base class.

The following illustration shows a class `WorkItem` that represents an item of work in some business process. Like all classes, it derives from System.Object and inherits all its methods. `WorkItem` adds six members of its own. These members include a constructor, because constructors aren't inherited. Class `ChangeRequest` inherits from `WorkItem` and represents a particular kind of work item. `ChangeRequest` adds two more members to the members that it inherits from `WorkItem` and from Object. It must add its own constructor, and it also adds `originalItemID`. Property `originalItemID` enables the

`ChangeRequest` instance to be associated with the original `WorkItem` to which the change request applies.



The following example shows how the class relationships demonstrated in the previous illustration are expressed in C#. The example also shows how `WorkItem` overrides the virtual method Object.ToString, and how the `ChangeRequest` class inherits the `WorkItem` implementation of the method. The first block defines the classes:

C#

```
// WorkItem implicitly inherits from the Object class.
public class WorkItem
{
    // Static field currentID stores the job ID of the last WorkItem that
    // has been created.
    private static int currentID;

    //Properties.
    protected int ID { get; set; }
    protected string Title { get; set; }
    protected string Description { get; set; }
    protected TimeSpan jobLength { get; set; }

    // Default constructor. If a derived class does not invoke a base-
    // class constructor explicitly, the default constructor is called
    // implicitly.
    public WorkItem()
    {
        ID = 0;
        Title = "Default title";
        Description = "Default description.";
        jobLength = new TimeSpan();
    }
```

```
    // Instance constructor that has three parameters.
    public WorkItem(string title, string desc, TimeSpan joblen)
    {
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
        this.jobLength = joblen;
    }

    // Static constructor to initialize the static member, currentID. This
    // constructor is called one time, automatically, before any instance
    // of WorkItem or ChangeRequest is created, or currentID is referenced.
    static WorkItem() => currentID = 0;

    // currentID is a static field. It is incremented each time a new
    // instance of WorkItem is created.
    protected int GetNextID() => ++currentID;

    // Method Update enables you to update the title and job length of an
    // existing WorkItem object.
    public void Update(string title, TimeSpan joblen)
    {
        this.Title = title;
        this.jobLength = joblen;
    }

    // Virtual method override of the ToString method that is inherited
    // from System.Object.
    public override string ToString() =>
        $"{this.ID} - {this.Title}";
}

// ChangeRequest derives from WorkItem and adds a property (originalItemID)
// and two constructors.
public class ChangeRequest : WorkItem
{
    protected int originalItemID { get; set; }

    // Constructors. Because neither constructor calls a base-class
    // constructor explicitly, the default constructor in the base class
    // is called implicitly. The base class must contain a default
    // constructor.

    // Default constructor for the derived class.
    public ChangeRequest() { }

    // Instance constructor that has four parameters.
    public ChangeRequest(string title, string desc, TimeSpan jobLen,
                         int originalID)
    {
        // The following properties and the GetNexID method are inherited
        // from WorkItem.
        this.ID = GetNextID();
        this.Title = title;
        this.Description = desc;
```

```
        this.jobLength = jobLen;

        // Property originalItemID is a member of ChangeRequest, but not
        // of WorkItem.
        this.originalItemID = originalID;
    }
}
```

This next block shows how to use the base and derived classes:

C#

```
// Create an instance of WorkItem by using the constructor in the
// base class that takes three arguments.
WorkItem item = new WorkItem("Fix Bugs",
                             "Fix all bugs in my code branch",
                             new TimeSpan(3, 4, 0, 0));

// Create an instance of ChangeRequest by using the constructor in
// the derived class that takes four arguments.
ChangeRequest change = new ChangeRequest("Change Base Class Design",
                                         "Add members to the class",
                                         new TimeSpan(4, 0, 0),
                                         1);

// Use the ToString method defined in WorkItem.
Console.WriteLine(item.ToString());

// Use the inherited Update method to change the title of the
// ChangeRequest object.
change.Update("Change the Design of the Base Class",
    new TimeSpan(4, 0, 0));

// ChangeRequest inherits WorkItem's override of ToString.
Console.WriteLine(change.ToString());
/* Output:
    1 - Fix Bugs
    2 - Change the Design of the Base Class
*/
```

Abstract and virtual methods

When a base class declares a method as virtual, a derived class can override the method with its own implementation. If a base class declares a member as abstract, that method must be overridden in any non-abstract class that directly inherits from that class. If a derived class is itself abstract, it inherits abstract members without implementing them. Abstract and virtual members are the basis for polymorphism, which is the second

primary characteristic of object-oriented programming. For more information, see Polymorphism.

Abstract base classes

You can declare a class as abstract if you want to prevent direct instantiation by using the new operator. An abstract class can be used only if a new class is derived from it. An abstract class can contain one or more method signatures that themselves are declared as abstract. These signatures specify the parameters and return value but have no implementation (method body). An abstract class doesn't have to contain abstract members; however, if a class does contain an abstract member, the class itself must be declared as abstract. Derived classes that aren't abstract themselves must provide the implementation for any abstract methods from an abstract base class.

Interfaces

An *interface* is a reference type that defines a set of members. All classes and structs that implement that interface must implement that set of members. An interface may define a default implementation for any or all of these members. A class can implement multiple interfaces even though it can derive from only a single direct base class.

Interfaces are used to define specific capabilities for classes that don't necessarily have an "is a" relationship. For example, the System.IEquatable<T> interface can be implemented by any class or struct to determine whether two objects of the type are equivalent (however the type defines equivalence). IEquatable<T> doesn't imply the same kind of "is a" relationship that exists between a base class and a derived class (for example, a `Mammal` is an `Animal`). For more information, see Interfaces.

Preventing further derivation

A class can prevent other classes from inheriting from it, or from any of its members, by declaring itself or the member as sealed.

Derived class hiding of base class members

A derived class can hide base class members by declaring members with the same name and signature. The new modifier can be used to explicitly indicate that the member isn't intended to be an override of the base member. The use of new isn't required, but a compiler warning will be generated if new isn't used. For more information, see

Versioning with the Override and New Keywords and Knowing When to Use Override and New Keywords.

Polymorphism

10/13/2025

Polymorphism is often referred to as the third pillar of object-oriented programming, after encapsulation and inheritance. Polymorphism is a Greek word that means "many-shaped" and it has two distinct aspects:

- At run time, objects of a derived class can be treated as objects of a base class in places such as method parameters and collections or arrays. When this polymorphism occurs, the object's declared type is no longer identical to its run-time type.
- Base classes might define and implement virtual *methods*, and derived classes can override them, which means they provide their own definition and implementation. At run-time, when client code calls the method, the CLR looks up the run-time type of the object, and invokes that override of the virtual method. In your source code you can call a method on a base class, and cause a derived class's version of the method to be executed.

Virtual methods enable you to work with groups of related objects in a uniform way. For example, suppose you have a drawing application that enables a user to create various kinds of shapes on a drawing surface. You don't know at compile time which specific types of shapes the user will create. However, the application has to keep track of all the various types of shapes that are created, and it has to update them in response to user mouse actions. You can use polymorphism to solve this problem in two basic steps:

1. Create a class hierarchy in which each specific shape class derives from a common base class.
2. Use a virtual method to invoke the appropriate method on any derived class through a single call to the base class method.

First, create a base class called `Shape`, and derived classes such as `Rectangle`, `Circle`, and `Triangle`. Give the `Shape` class a virtual method called `Draw`, and override it in each derived class to draw the particular shape that the class represents. Create a `List<Shape>` object and add a `Circle`, `Triangle`, and `Rectangle` to it.

C#

```
public class Shape
{
    // A few example members
    public int X { get; init; }
    public int Y { get; init; }
    public int Height { get; init; }
    public int Width { get; init; }
```

```
    // Virtual method
    public virtual void Draw()
    {
        Console.WriteLine("Performing base class drawing tasks");
    }
}

public class Circle : Shape
{
    public override void Draw()
    {
        // Code to draw a circle...
        Console.WriteLine("Drawing a circle");
        base.Draw();
    }
}
public class Rectangle : Shape
{
    public override void Draw()
    {
        // Code to draw a rectangle...
        Console.WriteLine("Drawing a rectangle");
        base.Draw();
    }
}
public class Triangle : Shape
{
    public override void Draw()
    {
        // Code to draw a triangle...
        Console.WriteLine("Drawing a triangle");
        base.Draw();
    }
}
```

To update the drawing surface, use a foreach loop to iterate through the list and call the `Draw` method on each `Shape` object in the list. Even though each object in the list has a declared type of `Shape`, it's the run-time type (the overridden version of the method in each derived class) that will be invoked.

C#

```
// Polymorphism at work #1: a Rectangle, Triangle and Circle
// can all be used wherever a Shape is expected. No cast is
// required because an implicit conversion exists from a derived
// class to its base class.
List<Shape> shapes =
[
    new Rectangle(),
    new Triangle(),
    new Circle()
];
```

```
// Polymorphism at work #2: the virtual method Draw is
// invoked on each of the derived classes, not the base class.
foreach (var shape in shapes)
{
    shape.Draw();
}
/* Output:
    Drawing a rectangle
    Performing base class drawing tasks
    Drawing a triangle
    Performing base class drawing tasks
    Drawing a circle
    Performing base class drawing tasks
*/
```

In C#, every type is polymorphic because all types, including user-defined types, inherit from Object.

Polymorphism overview

Virtual members

When a derived class inherits from a base class, it includes all the members of the base class. All the behavior declared in the base class is part of the derived class. That enables objects of the derived class to be treated as objects of the base class. Access modifiers (`public`, `protected`, `private`, and so on) determine if those members are accessible from the derived class implementation. Virtual methods give the designer different choices for the behavior of the derived class:

- The derived class can override virtual members in the base class, defining new behavior.
- The derived class can inherit the closest base class method without overriding it, preserving the existing behavior but enabling further derived classes to override the method.
- The derived class can define new non-virtual implementation of those members that hide the base class implementations.

A derived class can override a base class member only if the base class member is declared as virtual or abstract. The derived member must use the override keyword to explicitly indicate that the method is intended to participate in virtual invocation. The following code provides an example:

```
C#
```

```
public class BaseClass
{
    public virtual void DoWork() { }
    public virtual int WorkProperty => 0;
}
public class DerivedClass : BaseClass
{
    public override void DoWork() { }
    public override int WorkProperty
    {
        get { return 0; }
    }
}
```

Fields can't be virtual; only methods, properties, events, and indexers can be virtual. When a derived class overrides a virtual member, that member is called even when an instance of that class is being accessed as an instance of the base class. The following code provides an example:

C#

```
DerivedClass B = new();
B.DoWork();  // Calls the new method.

BaseClass A = B;
A.DoWork();  // Also calls the new method.
```

Virtual methods and properties enable derived classes to extend a base class without needing to use the base class implementation of a method. For more information, see Versioning with the Override and New Keywords. An interface provides another way to define a method or set of methods whose implementation is left to derived classes.

Hide base class members with new members

If you want your derived class to have a member with the same name as a member in a base class, you can use the new keyword to hide the base class member. The `new` keyword is put before the return type of a class member that's being replaced. The following code provides an example:

C#

```
public class BaseClass
{
    public void DoWork() { WorkField++; }
    public int WorkField;
    public int WorkProperty
```

```
    {
        get { return 0; }
    }
}

public class DerivedClass : BaseClass
{
    public new void DoWork() { WorkField++; }
    public new int WorkField;
    public new int WorkProperty
    {
        get { return 0; }
    }
}
```

When you use the `new` keyword, you're creating a method that *hides* the base class method rather than *overriding* it. This is different from virtual methods. With method hiding, the method that gets called depends on the compile-time type of the variable, not the run-time type of the object.

Hidden base class members can be accessed from client code by casting the instance of the derived class to an instance of the base class. For example:

C#

```
DerivedClass B = new();
B.DoWork();  // Calls the new method.

BaseClass A = (BaseClass)B;
A.DoWork();  // Calls the old method.
```

In this example, both variables refer to the same object instance, but the method that gets called depends on the variable's declared type: `DerivedClass.DoWork()` when accessed through the `DerivedClass` variable, and `BaseClass.DoWork()` when accessed through the `BaseClass` variable.

Prevent derived classes from overriding virtual members

Virtual members remain virtual, regardless of how many classes are declared between the virtual member and the class that originally declared it. If class `A` declares a virtual member, and class `B` derives from `A`, and class `C` derives from `B`, class `C` inherits the virtual member, and might override it, regardless of whether class `B` declared an override for that member. The following code provides an example:

C#

```
public class A
{
    public virtual void DoWork() { }
}
public class B : A
{
    public override void DoWork() { }
}
```

A derived class can stop virtual inheritance by declaring an override as sealed. Stopping inheritance requires putting the `sealed` keyword before the `override` keyword in the class member declaration. The following code provides an example:

C#

```
public class C : B
{
    public sealed override void DoWork() { }
}
```

In the previous example, the method `DoWork` is no longer virtual to any class derived from `C`. It's still virtual for instances of `C`, even if they're cast to type `B` or type `A`. Sealed methods can be replaced by derived classes by using the `new` keyword, as the following example shows:

C#

```
public class D : C
{
    public new void DoWork() { }
}
```

In this case, if `DoWork` is called on `D` using a variable of type `D`, the new `DoWork` is called. If a variable of type `C`, `B`, or `A` is used to access an instance of `D`, a call to `DoWork` follows the rules of virtual inheritance, routing those calls to the implementation of `DoWork` on class `C`.

Access base class virtual members from derived classes

A derived class that replaces or overrides a method or property can still access the method or property on the base class using the `base` keyword. The following code provides an example:

C#

```
public class Base
{
```

```
    public virtual void DoWork() {/*...*/ }
}
public class Derived : Base
{
    public override void DoWork()
    {
        //Perform Derived's work here
        //...
        // Call DoWork on base class
        base.DoWork();
    }
}
```

For more information, see base.

> **Note**
>
> It's recommended that virtual members use `base` to call the base class implementation of that member in their own implementation. Letting the base class behavior occur enables the derived class to concentrate on implementing behavior specific to the derived class. If the base class implementation isn't called, it's up to the derived class to make their behavior compatible with the behavior of the base class.

Pattern matching overview

Pattern matching is a technique where you test an expression to determine if it has certain characteristics. C# pattern matching provides more concise syntax for testing expressions and taking action when an expression matches. The "is expression" supports pattern matching to test an expression and conditionally declare a new variable to the result of that expression. The "switch expression" enables you to perform actions based on the first matching pattern for an expression. These two expressions support a rich vocabulary of *patterns*.

This article provides an overview of scenarios where you can use pattern matching. These techniques can improve the readability and correctness of your code. For a full discussion of all the patterns you can apply, see the article on patterns in the language reference.

Null checks

One of the most common scenarios for pattern matching is to ensure values aren't `null`. You can test and convert a nullable value type to its underlying type while testing for `null` using the following example:

C#

```
int? maybe = 12;

if (maybe is int number)
{
    Console.WriteLine($"The nullable int 'maybe' has the value {number}");
}
else
{
    Console.WriteLine("The nullable int 'maybe' doesn't hold a value");
}
```

The preceding code is a *declaration pattern* to test the type of the variable, and assign it to a new variable. The language rules make this technique safer than many others. The variable `number` is only accessible and assigned in the true portion of the `if` clause. If you try to access it elsewhere, either in the `else` clause, or after the `if` block, the compiler issues an error. Secondly, because you're not using the `==` operator, this pattern works when a type overloads the `==` operator. That makes it an ideal way to check null reference values, adding the `not` pattern:

C#

```
string? message = ReadMessageOrDefault();
```

```
if (message is not null)
{
    Console.WriteLine(message);
}
```

The preceding example used a *constant pattern* to compare the variable to `null`. The `not` is a *logical pattern* that matches when the negated pattern doesn't match.

Type tests

Another common use for pattern matching is to test a variable to see if it matches a given type. For example, the following code tests if a variable is non-null and implements the System.Collections.Generic.IList<T> interface. If it does, it uses the ICollection<T>.Count property on that list to find the middle index. The declaration pattern doesn't match a `null` value, regardless of the compile-time type of the variable. The code below guards against `null`, in addition to guarding against a type that doesn't implement `IList`.

C#

```
public static T MidPoint<T>(IEnumerable<T> sequence)
{
    if (sequence is IList<T> list)
    {
        return list[list.Count / 2];
    }
    else if (sequence is null)
    {
        throw new ArgumentNullException(nameof(sequence), "Sequence can't be
null.");
    }
    else
    {
        int halfLength = sequence.Count() / 2 - 1;
        if (halfLength < 0) halfLength = 0;
        return sequence.Skip(halfLength).First();
    }
}
```

The same tests can be applied in a `switch` expression to test a variable against multiple different types. You can use that information to create better algorithms based on the specific run-time type.

Compare discrete values

You can also test a variable to find a match on specific values. The following code shows one example where you test a value against all possible values declared in an enumeration:

C#

```
public State PerformOperation(Operation command) =>
   command switch
   {
       Operation.SystemTest => RunDiagnostics(),
       Operation.Start => StartSystem(),
       Operation.Stop => StopSystem(),
       Operation.Reset => ResetToReady(),
       _ => throw new ArgumentException("Invalid enum value for command",
nameof(command)),
   };
```

The previous example demonstrates a method dispatch based on the value of an enumeration. The final `_` case is a *discard pattern* that matches all values. It handles any error conditions where the value doesn't match one of the defined `enum` values. If you omit that switch arm, the compiler warns that your pattern expression doesn't handle all possible input values. At run time, the `switch` expression throws an exception if the object being examined doesn't match any of the switch arms. You could use numeric constants instead of a set of enum values. You can also use this similar technique for constant string values that represent the commands:

C#

```
public State PerformOperation(string command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command",
nameof(command)),
   };
```

The preceding example shows the same algorithm, but uses string values instead of an enum. You would use this scenario if your application responds to text commands instead of a regular data format. You can also use a `Span<char>` or a `ReadOnlySpan<char>` to test for constant string values, as shown in the following sample:

C#

```
public State PerformOperation(ReadOnlySpan<char> command) =>
   command switch
   {
       "SystemTest" => RunDiagnostics(),
       "Start" => StartSystem(),
       "Stop" => StopSystem(),
       "Reset" => ResetToReady(),
       _ => throw new ArgumentException("Invalid string value for command",
```

```
nameof(command)),
    };
```

In all these examples, the *discard pattern* ensures that you handle every input. The compiler helps you by making sure every possible input value is handled.

Relational patterns

You can use *relational patterns* to test how a value compares to constants. For example, the following code returns the state of water based on the temperature in Fahrenheit:

C#

```
string WaterState(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        (> 32) and (< 212) => "liquid",
        212 => "liquid / gas transition",
        > 212 => "gas",
    };
```

The preceding code also demonstrates the conjunctive `and` *logical pattern* to check that both relational patterns match. You can also use a disjunctive `or` pattern to check that either pattern matches. The two relational patterns are surrounded by parentheses, which you can use around any pattern for clarity. The two explicit switch arms (32°F and 212°F) handle the cases for the melting point and the boiling point. Without those two arms, the compiler warns you that your logic doesn't cover every possible input.

The preceding code also demonstrates another important feature the compiler provides for pattern matching expressions: The compiler warns you if you don't handle every input value. The compiler also issues a warning if the pattern for a switch arm is covered by a previous pattern. That gives you freedom to refactor and reorder switch expressions. Another way to write the same expression could be:

C#

```
string WaterState2(int tempInFahrenheit) =>
    tempInFahrenheit switch
    {
        < 32 => "solid",
        32 => "solid/liquid transition",
        < 212 => "liquid",
        212 => "liquid / gas transition",
```

```
        _ => "gas",
};
```

The key lesson in the preceding sample, and any other refactoring or reordering, is that the compiler validates that your code handles all possible inputs.

Multiple inputs

All the patterns covered so far have been checking one input. You can write patterns that examine multiple properties of an object. Consider the following `Order` record:

C#

```
public record Order(int Items, decimal Cost);
```

The preceding positional record type declares two members at explicit positions. Appearing first is the `Items`, then the order's `Cost`. For more information, see Records.

The following code examines the number of items and the value of an order to calculate a discounted price:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        { Items: > 10, Cost: > 1000.00m } => 0.10m,
        { Items: > 5, Cost: > 500.00m } => 0.05m,
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate
discount on null order"),
        var someObject => 0m,
    };
```

The first two arms examine two properties of the `Order`. The third examines only the cost. The next checks against `null`, and the final matches any other value. If the `Order` type defines a suitable Deconstruct method, you can omit the property names from the pattern and use deconstruction to examine properties:

C#

```
public decimal CalculateDiscount(Order order) =>
    order switch
    {
        ( > 10,  > 1000.00m) => 0.10m,
        ( > 5, > 50.00m) => 0.05m,
```

```
        { Cost: > 250.00m } => 0.02m,
        null => throw new ArgumentNullException(nameof(order), "Can't calculate
discount on null order"),
        var someObject => 0m,
    };
```

The preceding code demonstrates the *positional pattern* where the properties are deconstructed for the expression.

You can also match a property against `{ }`, which matches any non-null value. Consider the following declaration, which stores measurements with an optional annotation:

C#

```
public record class Observation(int Value, string Units, string Name)
{
    public string? Annotation { get; set; }
}
```

You can test if a given observation has a non-null annotation using the following pattern matching expression:

C#

```
if (observation.Annotation is { })
{
    Console.WriteLine($"Observation description: {observation.Annotation}");
}
```

List patterns

You can check elements in a list or an array using a *list pattern*. A list pattern provides a means to apply a pattern to any element of a sequence. In addition, you can apply the *discard pattern* (`_`) to match any element, or apply a *slice pattern* to match zero or more elements.

List patterns are a valuable tool when data doesn't follow a regular structure. You can use pattern matching to test the shape and values of the data instead of transforming it into a set of objects.

Consider the following excerpt from a text file containing bank transactions:

Output

```
04-01-2020, DEPOSIT,    Initial deposit,           2250.00
04-15-2020, DEPOSIT,    Refund,                     125.65
04-18-2020, DEPOSIT,    Paycheck,                   825.65
```

```
04-22-2020, WITHDRAWAL, Debit,           Groceries,  255.73
05-01-2020, WITHDRAWAL, #1102,           Rent, apt, 2100.00
05-02-2020, INTEREST,                                 0.65
05-07-2020, WITHDRAWAL, Debit,           Movies,      12.57
04-15-2020, FEE,                                      5.55
```

It's a CSV format, but some of the rows have more columns than others. Even worse for processing, one column in the `WITHDRAWAL` type contains user-generated text and can contain a comma in the text. A *list pattern* that includes the *discard* pattern, *constant* pattern, and *var* pattern to capture the value processes data in this format:

C#

```
decimal balance = 0m;
foreach (string[] transaction in ReadRecords())
{
    balance += transaction switch
    {
        [_, "DEPOSIT", _, var amount]     => decimal.Parse(amount),
        [_, "WITHDRAWAL", .., var amount] => -decimal.Parse(amount),
        [_, "INTEREST", var amount]       => decimal.Parse(amount),
        [_, "FEE", var fee]               => -decimal.Parse(fee),
        _                                 => throw new
InvalidOperationException($"Record {string.Join(", ", transaction)} is not in the
expected format!"),
    };
    Console.WriteLine($"Record: {string.Join(", ", transaction)}, New balance:
{balance:C}");
}
```

The preceding example takes a string array, where each element is one field in the row. The `switch` expression keys on the second field, which determines the kind of transaction, and the number of remaining columns. Each row ensures the data is in the correct format. The discard pattern (`_`) skips the first field, with the date of the transaction. The second field matches the type of transaction. Remaining element matches skip to the field with the amount. The final match uses the *var* pattern to capture the string representation of the amount. The expression calculates the amount to add or subtract from the balance.

List patterns enable you to match on the shape of a sequence of data elements. You use the *discard* and *slice* patterns to match the location of elements. You use other patterns to match characteristics about individual elements.

This article provided a tour of the kinds of code you can write with pattern matching in C#. The following articles show more examples of using patterns in scenarios, and the full vocabulary of patterns available to use.

See also

- Use pattern matching to avoid 'is' check followed by a cast (style rules IDE0020 and IDE0038)
- Exploration: Use pattern matching to build your class behavior for better code
- Tutorial: Use pattern matching to build type-driven and data-driven algorithms
- Reference: Pattern matching

Last updated on 11/18/2025

Discards - C# Fundamentals

Article • 02/19/2025

Discards are placeholder variables that are intentionally unused in application code. Discards are equivalent to unassigned variables; they don't have a value. A discard communicates intent to the compiler and others that read your code: You intended to ignore the result of an expression. You may want to ignore the result of an expression, one or more members of a tuple expression, an `out` parameter to a method, or the target of a pattern matching expression.

Discards make the intent of your code clear. A discard indicates that our code never uses the variable. They enhance its readability and maintainability.

You indicate that a variable is a discard by assigning it the underscore (`_`) as its name. For example, the following method call returns a tuple in which the first and second values are discards. `area` is a previously declared variable set to the third component returned by `GetCityInformation`:

```
C#

(_, _, area) = city.GetCityInformation(cityName);
```

You can use discards to specify unused input parameters of a lambda expression. For more information, see the Input parameters of a lambda expression section of the Lambda expressions article.

When `_` is a valid discard, attempting to retrieve its value or use it in an assignment operation generates compiler error CS0103, "The name '_' doesn't exist in the current context". This error is because `_` isn't assigned a value, and may not even be assigned a storage location. If it were an actual variable, you couldn't discard more than one value, as the previous example did.

Tuple and object deconstruction

Discards are useful in working with tuples when your application code uses some tuple elements but ignores others. For example, the following `QueryCityDataForYears` method returns a tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at

design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

C#

```
var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York City", 1960,
2010);

Console.WriteLine($"Population change, 1960 to 2010: {pop2 - pop1:N0}");

static (string, double, int, int, int, int) QueryCityDataForYears(string
name, int year1, int year2)
{
    int population1 = 0, population2 = 0;
    double area = 0;

    if (name == "New York City")
    {
        area = 468.48;
        if (year1 == 1960)
        {
            population1 = 7781984;
        }
        if (year2 == 2010)
        {
            population2 = 8175133;
        }
        return (name, area, year1, population1, year2, population2);
    }

    return ("", 0, 0, 0, 0, 0);
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

For more information on deconstructing tuples with discards, see Deconstructing tuples and other types.

The `Deconstruct` method of a class, structure, or interface also allows you to retrieve and deconstruct a specific set of data from an object. You can use discards when you're interested in working with only a subset of the deconstructed values. The following example deconstructs a `Person` object into four strings (the first and last names, the city, and the state), but discards the last name and the state.

C#

```
using System;

namespace Discards
{
```

```
    public class Person
    {
        public string FirstName { get; set; }
        public string MiddleName { get; set; }
        public string LastName { get; set; }
        public string City { get; set; }
        public string State { get; set; }

        public Person(string fname, string mname, string lname,
                      string cityName, string stateName)
        {
            FirstName = fname;
            MiddleName = mname;
            LastName = lname;
            City = cityName;
            State = stateName;
        }

        // Return the first and last name.
        public void Deconstruct(out string fname, out string lname)
        {
            fname = FirstName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string mname, out
string lname)
        {
            fname = FirstName;
            mname = MiddleName;
            lname = LastName;
        }

        public void Deconstruct(out string fname, out string lname,
                                out string city, out string state)
        {
            fname = FirstName;
            lname = LastName;
            city = City;
            state = State;
        }
    }
    class Example
    {
        public static void Main()
        {
            var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

            // Deconstruct the person object.
            var (fName, _, city, _) = p;
            Console.WriteLine($"Hello {fName} of {city}!");
            // The example displays the following output:
            //      Hello John of Boston!
        }
```

```
    }
}
```

For more information on deconstructing user-defined types with discards, see Deconstructing tuples and other types.

Pattern matching with `switch`

The *discard pattern* can be used in pattern matching with the switch expression. Every expression, including `null`, always matches the discard pattern.

The following example defines a `ProvidesFormatInfo` method that uses a `switch` expression to determine whether an object provides an IFormatProvider implementation and tests whether the object is `null`. It also uses the discard pattern to handle non-null objects of any other type.

C#

```
object?[] objects = [CultureInfo.CurrentCulture,
                     CultureInfo.CurrentCulture.DateTimeFormat,
                     CultureInfo.CurrentCulture.NumberFormat,
                     new ArgumentException(), null];
foreach (var obj in objects)
    ProvidesFormatInfo(obj);

static void ProvidesFormatInfo(object? obj) =>
    Console.WriteLine(obj switch
    {
        IFormatProvider fmt => $"{fmt.GetType()} object",
        null => "A null object reference: Its use could result in a
NullReferenceException",
        _ => "Some object type without format information"
    });
// The example displays the following output:
//    System.Globalization.CultureInfo object
//    System.Globalization.DateTimeFormatInfo object
//    System.Globalization.NumberFormatInfo object
//    Some object type without format information
//    A null object reference: Its use could result in a
NullReferenceException
```

Calls to methods with `out` parameters

When calling the `Deconstruct` method to deconstruct a user-defined type (an instance of a class, structure, or interface), you can discard the values of individual `out`

arguments. But you can also discard the value of `out` arguments when calling any method with an `out` parameter.

The following example calls the DateTime.TryParse(String, out DateTime) method to determine whether the string representation of a date is valid in the current culture. Because the example is concerned only with validating the date string and not with parsing it to extract the date, the `out` argument to the method is a discard.

C#

```
string[] dateStrings = ["05/01/2018 14:57:32.8", "2018-05-01 14:57:32.8",
                        "2018-05-01T14:57:32.8375298-04:00", "5/01/2018",
                        "5/01/2018 14:57:32.80 -07:00",
                        "1 May 2018 2:57:32.8 PM", "16-05-2018 1:00:32 PM",
                        "Fri, 15 May 2018 20:10:57 GMT"];
foreach (string dateString in dateStrings)
{
    if (DateTime.TryParse(dateString, out _))
        Console.WriteLine($"'{dateString}': valid");
    else
        Console.WriteLine($"'{dateString}': invalid");
}
// The example displays output like the following:
//       '05/01/2018 14:57:32.8': valid
//       '2018-05-01 14:57:32.8': valid
//       '2018-05-01T14:57:32.8375298-04:00': valid
//       '5/01/2018': valid
//       '5/01/2018 14:57:32.80 -07:00': valid
//       '1 May 2018 2:57:32.8 PM': valid
//       '16-05-2018 1:00:32 PM': invalid
//       'Fri, 15 May 2018 20:10:57 GMT': invalid
```

A standalone discard

You can use a standalone discard to indicate any variable that you choose to ignore. One typical use is to use an assignment to ensure that an argument isn't null. The following code uses a discard to force an assignment. The right side of the assignment uses the null coalescing operator to throw an System.ArgumentNullException when the argument is `null`. The code doesn't need the result of the assignment, so it's discarded. The expression forces a null check. The discard clarifies your intent: the result of the assignment isn't needed or used.

C#

```
public static void Method(string arg)
{
    _ = arg ?? throw new ArgumentNullException(paramName: nameof(arg),
```

```
message: "arg can't be null");

    // Do work with arg.
}
```

The following example uses a standalone discard to ignore the Task object returned by an asynchronous operation. Assigning the task has the effect of suppressing the exception that the operation throws as it is about to complete. It makes your intent clear: You want to discard the `Task`, and ignore any errors generated from that asynchronous operation.

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    _ = Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
    });
    await Task.Delay(5000);
    Console.WriteLine("Exiting after 5 second delay");
}
// The example displays output like the following:
//       About to launch a task...
//       Completed looping operation...
//       Exiting after 5 second delay
```

Without assigning the task to a discard, the following code generates a compiler warning:

C#

```
private static async Task ExecuteAsyncMethods()
{
    Console.WriteLine("About to launch a task...");
    // CS4014: Because this call is not awaited, execution of the current
method continues before the call is completed.
    // Consider applying the 'await' operator to the result of the call.
    Task.Run(() =>
    {
        var iterations = 0;
        for (int ctr = 0; ctr < int.MaxValue; ctr++)
            iterations++;
        Console.WriteLine("Completed looping operation...");
        throw new InvalidOperationException();
```

```
});
await Task.Delay(5000);
Console.WriteLine("Exiting after 5 second delay");
```

> **Note**
>
> If you run either of the preceding two samples using a debugger, the debugger will stop the program when the exception is thrown. Without a debugger attached, the exception is silently ignored in both cases.

`_` is also a valid identifier. When used outside of a supported context, `_` is treated not as a discard but as a valid variable. If an identifier named `_` is already in scope, the use of `_` as a standalone discard can result in:

- Accidental modification of the value of the in-scope `_` variable by assigning it the value of the intended discard. For example:

  C#

  ```
  private static void ShowValue(int _)
  {
     byte[] arr = [0, 0, 1, 2];
     _ = BitConverter.ToInt32(arr, 0);
     Console.WriteLine(_);
  }
   // The example displays the following output:
   //       33619968
  ```

- A compiler error for violating type safety. For example:

  C#

  ```
  private static bool RoundTrips(int _)
  {
     string value = _.ToString();
     int newValue = 0;
     _ = Int32.TryParse(value, out newValue);
     return _ == newValue;
  }
  // The example displays the following compiler error:
  //      error CS0029: Cannot implicitly convert type 'bool' to 'int'
  ```

See also

- Remove unnecessary expression value (style rule IDE0058)
- Remove unnecessary value assignment (style rule IDE0059)
- Remove unused parameter (style rule IDE0060)
- Deconstructing tuples and other types
- is operator
- switch expression

Deconstructing tuples and other types

Article • 12/04/2024

A tuple provides a lightweight way to retrieve multiple values from a method call. But once you retrieve the tuple, you have to handle its individual elements. Working on an element-by-element basis is cumbersome, as the following example shows. The `QueryCityData` method returns a three-tuple, and each of its elements is assigned to a variable in a separate operation.

C#

```
public class Example
{
    public static void Main()
    {
        var result = QueryCityData("New York City");

        var city = result.Item1;
        var pop = result.Item2;
        var size = result.Item3;

         // Do something with the data.
    }

    private static (string, int, double) QueryCityData(string name)
    {
        if (name == "New York City")
            return (name, 8175133, 468.48);

        return ("", 0, 0);
    }
}
```

Retrieving multiple field and property values from an object can be equally cumbersome: you must assign a field or property value to a variable on a member-by-member basis.

You can retrieve multiple elements from a tuple or retrieve multiple field, property, and computed values from an object in a single *deconstruct* operation. To deconstruct a tuple, you assign its elements to individual variables. When you deconstruct an object, you assign selected values to individual variables.

Tuples

C# features built-in support for deconstructing tuples, which lets you unpackage all the items in a tuple in a single operation. The general syntax for deconstructing a tuple is similar to the syntax for defining one: you enclose the variables to which each element is to be assigned in parentheses in the left side of an assignment statement. For example, the following statement assigns the elements of a four-tuple to four separate variables:

```
C#

var (name, address, city, zip) = contact.GetAddressInfo();
```

There are three ways to deconstruct a tuple:

- You can explicitly declare the type of each field inside parentheses. The following example uses this approach to deconstruct the three-tuple returned by the `QueryCityData` method.

  ```
  C#

  public static void Main()
  {
      (string city, int population, double area) = QueryCityData("New
  York City");

      // Do something with the data.
  }
  ```

- You can use the `var` keyword so that C# infers the type of each variable. You place the `var` keyword outside of the parentheses. The following example uses type inference when deconstructing the three-tuple returned by the `QueryCityData` method.

  ```
  C#

  public static void Main()
  {
      var (city, population, area) = QueryCityData("New York City");

      // Do something with the data.
  }
  ```

  You can also use the `var` keyword individually with any or all of the variable declarations inside the parentheses.

  ```
  C#
  ```

```
public static void Main()
{
    (string city, var population, var area) = QueryCityData("New York
City");

    // Do something with the data.
}
```

The preceding example is cumbersome and isn't recommended.

- Lastly, you can deconstruct the tuple into variables already declared.

```
C#
```

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;
    double area = 144.8;

    (city, population, area) = QueryCityData("New York City");

    // Do something with the data.
}
```

- You can mix variable declaration and assignment in a deconstruction.

```
C#
```

```
public static void Main()
{
    string city = "Raleigh";
    int population = 458880;

    (city, population, double area) = QueryCityData("New York City");

    // Do something with the data.
}
```

You can't specify a specific type outside the parentheses even if every field in the tuple has the same type. Doing so generates compiler error CS8136, "Deconstruction 'var (...)' form disallows a specific type for 'var'."

You must assign each element of the tuple to a variable. If you omit any elements, the compiler generates error CS8132, "Can't deconstruct a tuple of 'x' elements into 'y' variables."

Tuple elements with discards

Often when deconstructing a tuple, you're interested in the values of only some elements. You can take advantage of C#'s support for *discards*, which are write-only variables whose values you chose to ignore. You declare a discard with an underscore character ("_") in an assignment. You can discard as many values as you like; a single discard, `_`, represents all the discarded values.

The following example illustrates the use of tuples with discards. The `QueryCityDataForYears` method returns a six-tuple with the name of a city, its area, a year, the city's population for that year, a second year, and the city's population for that second year. The example shows the change in population between those two years. Of the data available from the tuple, we're unconcerned with the city area, and we know the city name and the two dates at design-time. As a result, we're only interested in the two population values stored in the tuple, and can handle its remaining values as discards.

C#

```
using System;

public class ExampleDiscard
{
    public static void Main()
    {
        var (_, _, _, pop1, _, pop2) = QueryCityDataForYears("New York
City", 1960, 2010);

        Console.WriteLine($"Population change, 1960 to 2010: {pop2 -
pop1:N0}");
    }

    private static (string, double, int, int, int, int)
QueryCityDataForYears(string name, int year1, int year2)
    {
        int population1 = 0, population2 = 0;
        double area = 0;

        if (name == "New York City")
        {
            area = 468.48;
            if (year1 == 1960)
            {
                population1 = 7781984;
            }
            if (year2 == 2010)
            {
                population2 = 8175133;
            }
```

```
            return (name, area, year1, population1, year2, population2);
        }

        return ("", 0, 0, 0, 0, 0);
    }
}
// The example displays the following output:
//      Population change, 1960 to 2010: 393,149
```

User-defined types

C# offers built-in support for deconstructing tuple types, record, and DictionaryEntry types. However, as the author of a class, a struct, or an interface, you can allow instances of the type to be deconstructed by implementing one or more `Deconstruct` methods. The method returns void. An out parameter in the method signature represents each value to be deconstructed. For example, the following `Deconstruct` method of a `Person` class returns the first, middle, and family name:

C#

```
public void Deconstruct(out string fname, out string mname, out string
lname)
```

You can then deconstruct an instance of the `Person` class named `p` with an assignment like the following code:

C#

```
var (fName, mName, lName) = p;
```

The following example overloads the `Deconstruct` method to return various combinations of properties of a `Person` object. Individual overloads return:

- A first and family name.
- A first, middle, and family name.
- A first name, a family name, a city name, and a state name.

C#

```
using System;

public class Person
{
    public string FirstName { get; set; }
    public string MiddleName { get; set; }
```

```
    public string LastName { get; set; }
    public string City { get; set; }
    public string State { get; set; }

    public Person(string fname, string mname, string lname,
                  string cityName, string stateName)
    {
        FirstName = fname;
        MiddleName = mname;
        LastName = lname;
        City = cityName;
        State = stateName;
    }

    // Return the first and last name.
    public void Deconstruct(out string fname, out string lname)
    {
        fname = FirstName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string mname, out string
lname)
    {
        fname = FirstName;
        mname = MiddleName;
        lname = LastName;
    }

    public void Deconstruct(out string fname, out string lname,
                            out string city, out string state)
    {
        fname = FirstName;
        lname = LastName;
        city = City;
        state = State;
    }
}

public class ExampleClassDeconstruction
{
    public static void Main()
    {
        var p = new Person("John", "Quincy", "Adams", "Boston", "MA");

        // Deconstruct the person object.
        var (fName, lName, city, state) = p;
        Console.WriteLine($"Hello {fName} {lName} of {city}, {state}!");
    }
}
// The example displays the following output:
//    Hello John Adams of Boston, MA!
```

Multiple `Deconstruct` methods having the same number of parameters are ambiguous. You must be careful to define `Deconstruct` methods with different numbers of parameters, or "arity". `Deconstruct` methods with the same number of parameters can't be distinguished during overload resolution.

User-defined type with discards

Just as you do with tuples, you can use discards to ignore selected items returned by a `Deconstruct` method. A variable named "_" represents a discard. A single deconstruction operation can include multiple discards.

The following example deconstructs a `Person` object into four strings (the first and family names, the city, and the state) but discards the family name and the state.

C#

```
// Deconstruct the person object.
var (fName, _, city, _) = p;
Console.WriteLine($"Hello {fName} of {city}!");
// The example displays the following output:
//      Hello John of Boston!
```

Deconstruction extension methods

If you didn't author a class, struct, or interface, you can still deconstruct objects of that type by implementing one or more `Deconstruct` extension methods to return the values in which you're interested.

The following example defines two `Deconstruct` extension methods for the System.Reflection.PropertyInfo class. The first returns a set of values that indicate the characteristics of the property. The second indicates the property's accessibility. Boolean values indicate whether the property has separate get and set accessors or different accessibility. If there's only one accessor or both the get and the set accessor have the same accessibility, the `access` variable indicates the accessibility of the property as a whole. Otherwise, the accessibility of the get and set accessors are indicated by the `getAccess` and `setAccess` variables.

C#

```
using System;
using System.Collections.Generic;
using System.Reflection;
```

```
public static class ReflectionExtensions
{
    public static void Deconstruct(this PropertyInfo p, out bool isStatic,
                                   out bool isReadOnly, out bool isIndexed,
                                   out Type propertyType)
    {
        var getter = p.GetMethod;

        // Is the property read-only?
        isReadOnly = ! p.CanWrite;

        // Is the property instance or static?
        isStatic = getter.IsStatic;

        // Is the property indexed?
        isIndexed = p.GetIndexParameters().Length > 0;

        // Get the property type.
        propertyType = p.PropertyType;
    }

    public static void Deconstruct(this PropertyInfo p, out bool
hasGetAndSet,
                                   out bool sameAccess, out string access,
                                   out string getAccess, out string
setAccess)
    {
        hasGetAndSet = sameAccess = false;
        string getAccessTemp = null;
        string setAccessTemp = null;

        MethodInfo getter = null;
        if (p.CanRead)
            getter = p.GetMethod;

        MethodInfo setter = null;
        if (p.CanWrite)
            setter = p.SetMethod;

        if (setter != null && getter != null)
            hasGetAndSet = true;

        if (getter != null)
        {
            if (getter.IsPublic)
                getAccessTemp = "public";
            else if (getter.IsPrivate)
                getAccessTemp = "private";
            else if (getter.IsAssembly)
                getAccessTemp = "internal";
            else if (getter.IsFamily)
                getAccessTemp = "protected";
            else if (getter.IsFamilyOrAssembly)
                getAccessTemp = "protected internal";
```

```
        }

        if (setter != null)
        {
            if (setter.IsPublic)
                setAccessTemp = "public";
            else if (setter.IsPrivate)
                setAccessTemp = "private";
            else if (setter.IsAssembly)
                setAccessTemp = "internal";
            else if (setter.IsFamily)
                setAccessTemp = "protected";
            else if (setter.IsFamilyOrAssembly)
                setAccessTemp = "protected internal";
        }

        // Are the accessibility of the getter and setter the same?
        if (setAccessTemp == getAccessTemp)
        {
            sameAccess = true;
            access = getAccessTemp;
            getAccess = setAccess = String.Empty;
        }
        else
        {
            access = null;
            getAccess = getAccessTemp;
            setAccess = setAccessTemp;
        }
    }
}

public class ExampleExtension
{
    public static void Main()
    {
        Type dateType = typeof(DateTime);
        PropertyInfo prop = dateType.GetProperty("Now");
        var (isStatic, isRO, isIndexed, propType) = prop;
        Console.WriteLine($"\nThe {dateType.FullName}.{prop.Name}
property:");
        Console.WriteLine($"   PropertyType: {propType.Name}");
        Console.WriteLine($"   Static:       {isStatic}");
        Console.WriteLine($"   Read-only:    {isRO}");
        Console.WriteLine($"   Indexed:      {isIndexed}");

        Type listType = typeof(List<>);
        prop = listType.GetProperty("Item",
                                    BindingFlags.Public |
BindingFlags.NonPublic | BindingFlags.Instance | BindingFlags.Static);
        var (hasGetAndSet, sameAccess, accessibility, getAccessibility,
setAccessibility) = prop;
        Console.Write($"\nAccessibility of the {listType.FullName}.
{prop.Name} property: ");
```

```
        if (!hasGetAndSet | sameAccess)
        {
            Console.WriteLine(accessibility);
        }
        else
        {
            Console.WriteLine($"\n   The get accessor: {getAccessibility}");
            Console.WriteLine($"   The set accessor: {setAccessibility}");
        }
    }
}
// The example displays the following output:
//       The System.DateTime.Now property:
//          PropertyType: DateTime
//          Static:       True
//          Read-only:    True
//          Indexed:      False
//
//       Accessibility of the System.Collections.Generic.List`1.Item
property: public
```

Extension method for system types

Some system types provide the `Deconstruct` method as a convenience. For example, the System.Collections.Generic.KeyValuePair<TKey,TValue> type provides this functionality. When you're iterating over a System.Collections.Generic.Dictionary<TKey,TValue>, each element is a `KeyValuePair<TKey, TValue>` and can be deconstructed. Consider the following example:

C#

```
Dictionary<string, int> snapshotCommitMap =
new(StringComparer.OrdinalIgnoreCase)
{
    ["https://github.com/dotnet/docs"] = 16_465,
    ["https://github.com/dotnet/runtime"] = 114_223,
    ["https://github.com/dotnet/installer"] = 22_436,
    ["https://github.com/dotnet/roslyn"] = 79_484,
    ["https://github.com/dotnet/aspnetcore"] = 48_386
};

foreach (var (repo, commitCount) in snapshotCommitMap)
{
    Console.WriteLine(
        $"The {repo} repository had {commitCount:N0} commits as of November
10th, 2021.");
}
```

`record` types

When you declare a record type by using two or more positional parameters, the compiler creates a `Deconstruct` method with an `out` parameter for each positional parameter in the `record` declaration. For more information, see Positional syntax for property definition and Deconstructor behavior in derived records.

See also

- Deconstruct variable declaration (style rule IDE0042)
- Discards
- Tuple types

Exceptions and Exception Handling

Article • 04/22/2023

The C# language's exception handling features help you deal with any unexpected or exceptional situations that occur when a program is running. Exception handling uses the `try`, `catch`, and `finally` keywords to try actions that may not succeed, to handle failures when you decide that it's reasonable to do so, and to clean up resources afterward. Exceptions can be generated by the common language runtime (CLR), by .NET or third-party libraries, or by application code. Exceptions are created by using the `throw` keyword.

In many cases, an exception may be thrown not by a method that your code has called directly, but by another method further down in the call stack. When an exception is thrown, the CLR will unwind the stack, looking for a method with a `catch` block for the specific exception type, and it will execute the first such `catch` block that it finds. If it finds no appropriate `catch` block anywhere in the call stack, it will terminate the process and display a message to the user.

In this example, a method tests for division by zero and catches the error. Without the exception handling, this program would terminate with a **DivideByZeroException was unhandled** error.

C#

```
public class ExceptionTest
{
    static double SafeDivision(double x, double y)
    {
        if (y == 0)
            throw new DivideByZeroException();
        return x / y;
    }

    public static void Main()
    {
        // Input for test purposes. Change the values to see
        // exception handling behavior.
        double a = 98, b = 0;
        double result;

        try
        {
            result = SafeDivision(a, b);
            Console.WriteLine($"{a} divided by {b} = {result}");
        }
        catch (DivideByZeroException)
```

```
        {
            Console.WriteLine("Attempted divide by zero.");
        }
    }
}
```

Exceptions Overview

Exceptions have the following properties:

- Exceptions are types that all ultimately derive from `System.Exception`.
- Use a `try` block around the statements that might throw exceptions.
- Once an exception occurs in the `try` block, the flow of control jumps to the first associated exception handler that is present anywhere in the call stack. In C#, the `catch` keyword is used to define an exception handler.
- If no exception handler for a given exception is present, the program stops executing with an error message.
- Don't catch an exception unless you can handle it and leave the application in a known state. If you catch `System.Exception`, rethrow it using the `throw` keyword at the end of the `catch` block.
- If a `catch` block defines an exception variable, you can use it to obtain more information about the type of exception that occurred.
- Exceptions can be explicitly generated by a program by using the `throw` keyword.
- Exception objects contain detailed information about the error, such as the state of the call stack and a text description of the error.
- Code in a `finally` block is executed regardless of if an exception is thrown. Use a `finally` block to release resources, for example to close any streams or files that were opened in the `try` block.
- Managed exceptions in .NET are implemented on top of the Win32 structured exception handling mechanism. For more information, see Structured Exception Handling (C/C++) and A Crash Course on the Depths of Win32 Structured Exception Handling.

C# Language Specification

For more information, see Exceptions in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- System.Exception
- Exception-handling statements
- Exceptions

Use exceptions

Article • 09/15/2021

In C#, errors in the program at run time are propagated through the program by using a mechanism called exceptions. Exceptions are thrown by code that encounters an error and caught by code that can correct the error. Exceptions can be thrown by the .NET runtime or by code in a program. Once an exception is thrown, it propagates up the call stack until a `catch` statement for the exception is found. Uncaught exceptions are handled by a generic exception handler provided by the system that displays a dialog box.

Exceptions are represented by classes derived from Exception. This class identifies the type of exception and contains properties that have details about the exception. Throwing an exception involves creating an instance of an exception-derived class, optionally configuring properties of the exception, and then throwing the object by using the `throw` keyword. For example:

C#

```
class CustomException : Exception
{
    public CustomException(string message)
    {
    }
}
private static void TestThrow()
{
    throw new CustomException("Custom exception in TestThrow()");
}
```

After an exception is thrown, the runtime checks the current statement to see whether it is within a `try` block. If it is, any `catch` blocks associated with the `try` block are checked to see whether they can catch the exception. `Catch` blocks typically specify exception types; if the type of the `catch` block is the same type as the exception, or a base class of the exception, the `catch` block can handle the method. For example:

C#

```
try
{
    TestThrow();
}
catch (CustomException ex)
{
```

```
    System.Console.WriteLine(ex.ToString());
}
```

If the statement that throws an exception isn't within a `try` block or if the `try` block that encloses it has no matching `catch` block, the runtime checks the calling method for a `try` statement and `catch` blocks. The runtime continues up the calling stack, searching for a compatible `catch` block. After the `catch` block is found and executed, control is passed to the next statement after that `catch` block.

A `try` statement can contain more than one `catch` block. The first `catch` statement that can handle the exception is executed; any following `catch` statements, even if they're compatible, are ignored. Order catch blocks from most specific (or most-derived) to least specific. For example:

C#

```
using System;
using System.IO;

namespace Exceptions
{
    public class CatchOrder
    {
        public static void Main()
        {
            try
            {
                using (var sw = new StreamWriter("./test.txt"))
                {
                    sw.WriteLine("Hello");
                }
            }
            // Put the more specific exceptions first.
            catch (DirectoryNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            catch (FileNotFoundException ex)
            {
                Console.WriteLine(ex);
            }
            // Put the least specific exception last.
            catch (IOException ex)
            {
                Console.WriteLine(ex);
            }
            Console.WriteLine("Done");
        }
```

```
    }
}
```

Before the `catch` block is executed, the runtime checks for `finally` blocks. `Finally` blocks enable the programmer to clean up any ambiguous state that could be left over from an aborted `try` block, or to release any external resources (such as graphics handles, database connections, or file streams) without waiting for the garbage collector in the runtime to finalize the objects. For example:

C#

```
static void TestFinally()
{
    FileStream? file = null;
    //Change the path to something that works on your machine.
    FileInfo fileInfo = new System.IO.FileInfo("./file.txt");

    try
    {
        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    finally
    {
        // Closing the file allows you to reopen it immediately - otherwise
IOException is thrown.
        file?.Close();
    }

    try
    {
        file = fileInfo.OpenWrite();
        Console.WriteLine("OpenWrite() succeeded");
    }
    catch (IOException)
    {
        Console.WriteLine("OpenWrite() failed");
    }
}
```

If `WriteByte()` threw an exception, the code in the second `try` block that tries to reopen the file would fail if `file.Close()` isn't called, and the file would remain locked. Because `finally` blocks are executed even if an exception is thrown, the `finally` block in the previous example allows for the file to be closed correctly and helps avoid an error.

If no compatible `catch` block is found on the call stack after an exception is thrown, one of three things occurs:

- If the exception is within a finalizer, the finalizer is aborted and the base finalizer, if any, is called.
- If the call stack contains a static constructor, or a static field initializer, a TypeInitializationException is thrown, with the original exception assigned to the InnerException property of the new exception.
- If the start of the thread is reached, the thread is terminated.

Exception Handling (C# Programming Guide)

Article • 03/14/2023

A try block is used by C# programmers to partition code that might be affected by an exception. Associated catch blocks are used to handle any resulting exceptions. A finally block contains code that is run whether or not an exception is thrown in the `try` block, such as releasing resources that are allocated in the `try` block. A `try` block requires one or more associated `catch` blocks, or a `finally` block, or both.

The following examples show a `try-catch` statement, a `try-finally` statement, and a `try-catch-finally` statement.

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
{
    // Code to handle the exception goes here.
    // Only catch exceptions that you know how to handle.
    // Never catch base class System.Exception without
    // rethrowing it at the end of the catch block.
}
```

C#

```
try
{
    // Code to try goes here.
}
finally
{
    // Code to execute after the try block goes here.
}
```

C#

```
try
{
    // Code to try goes here.
}
catch (SomeSpecificException ex)
```

```
{
    // Code to handle the exception goes here.
}
finally
{
    // Code to execute after the try (and possibly catch) blocks
    // goes here.
}
```

A `try` block without a `catch` or `finally` block causes a compiler error.

Catch Blocks

A `catch` block can specify the type of exception to catch. The type specification is called an *exception filter*. The exception type should be derived from Exception. In general, don't specify Exception as the exception filter unless either you know how to handle all exceptions that might be thrown in the `try` block, or you've included a throw statement at the end of your `catch` block.

Multiple `catch` blocks with different exception classes can be chained together. The `catch` blocks are evaluated from top to bottom in your code, but only one `catch` block is executed for each exception that is thrown. The first `catch` block that specifies the exact type or a base class of the thrown exception is executed. If no `catch` block specifies a matching exception class, a `catch` block that doesn't have any type is selected, if one is present in the statement. It's important to position `catch` blocks with the most specific (that is, the most derived) exception classes first.

Catch exceptions when the following conditions are true:

- You have a good understanding of why the exception might be thrown, and you can implement a specific recovery, such as prompting the user to enter a new file name when you catch a FileNotFoundException object.
- You can create and throw a new, more specific exception.

C#

```
int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
```

```
                "Parameter index is out of range.", e);
        }
    }
```

- You want to partially handle an exception before passing it on for more handling. In the following example, a `catch` block is used to add an entry to an error log before rethrowing the exception.

```
C#

try
{
    // Try to access a resource.
}
catch (UnauthorizedAccessException e)
{
    // Call a custom error logging procedure.
    LogError(e);
    // Re-throw the error.
    throw;
}
```

You can also specify *exception filters* to add a boolean expression to a catch clause. Exception filters indicate that a specific catch clause matches only when that condition is true. In the following example, both catch clauses use the same exception class, but an extra condition is checked to create a different error message:

```
C#

int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e) when (index < 0)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be negative.", e);
    }
    catch (IndexOutOfRangeException e)
    {
        throw new ArgumentOutOfRangeException(
            "Parameter index cannot be greater than the array size.", e);
    }
}
```

An exception filter that always returns `false` can be used to examine all exceptions but not process them. A typical use is to log exceptions:

C#

```
public class ExceptionFilter
{
    public static void Main()
    {
        try
        {
            string? s = null;
            Console.WriteLine(s.Length);
        }
        catch (Exception e) when (LogException(e))
        {
        }
        Console.WriteLine("Exception must have been handled");
    }

    private static bool LogException(Exception e)
    {
        Console.WriteLine($"\tIn the log routine. Caught {e.GetType()}");
        Console.WriteLine($"\tMessage: {e.Message}");
        return false;
    }
}
```

The `LogException` method always returns `false`, no `catch` clause using this exception filter matches. The catch clause can be general, using System.Exception, and later clauses can process more specific exception classes.

Finally Blocks

A `finally` block enables you to clean up actions that are performed in a `try` block. If present, the `finally` block executes last, after the `try` block and any matched `catch` block. A `finally` block always runs, whether an exception is thrown or a `catch` block matching the exception type is found.

The `finally` block can be used to release resources such as file streams, database connections, and graphics handles without waiting for the garbage collector in the runtime to finalize the objects.

In the following example, the `finally` block is used to close a file that is opened in the `try` block. Notice that the state of the file handle is checked before the file is closed. If the `try` block can't open the file, the file handle still has the value `null` and the `finally` block doesn't try to close it. Instead, if the file is opened successfully in the `try` block, the `finally` block closes the open file.

C#

```
FileStream? file = null;
FileInfo fileinfo = new System.IO.FileInfo("./file.txt");
try
{
    file = fileinfo.OpenWrite();
    file.WriteByte(0xF);
}
finally
{
    // Check for null because OpenWrite might have failed.
    file?.Close();
}
```

C# Language Specification

For more information, see Exceptions and The try statement in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# reference
- Exception-handling statements
- using statement

Create and throw exceptions

Article • 11/03/2023

Exceptions are used to indicate that an error has occurred while running the program. Exception objects that describe an error are created and then *thrown* with the throw statement or expression. The runtime then searches for the most compatible exception handler.

Programmers should throw exceptions when one or more of the following conditions are true:

- The method can't complete its defined functionality. For example, if a parameter to a method has an invalid value:

  C#

  ```
  static void CopyObject(SampleClass original)
  {
      _ = original ?? throw new ArgumentException("Parameter cannot be
  null", nameof(original));
  }
  ```

- An inappropriate call to an object is made, based on the object state. One example might be trying to write to a read-only file. In cases where an object state doesn't allow an operation, throw an instance of InvalidOperationException or an object based on a derivation of this class. The following code is an example of a method that throws an InvalidOperationException object:

  C#

  ```
  public class ProgramLog
  {
      FileStream logFile = null!;
      public void OpenLog(FileInfo fileName, FileMode mode) { }

      public void WriteLog()
      {
          if (!logFile.CanWrite)
          {
              throw new InvalidOperationException("Logfile cannot be
  read-only");
          }
          // Else write data to the log and return.
      }
  }
  ```

- When an argument to a method causes an exception. In this case, the original exception should be caught and an ArgumentException instance should be created. The original exception should be passed to the constructor of the ArgumentException as the InnerException parameter:

  C#

  ```
  static int GetValueFromArray(int[] array, int index)
  {
      try
      {
          return array[index];
      }
      catch (IndexOutOfRangeException e)
      {
          throw new ArgumentOutOfRangeException(
              "Parameter index is out of range.", e);
      }
  }
  ```

  > **Note**
  >
  > The preceding example shows how to use the `InnerException` property. It's intentionally simplified. In practice, you should check that an index is in range before using it. You could use this technique of wrapping an exception when a member of a parameter throws an exception you couldn't anticipate before calling the member.

Exceptions contain a property named StackTrace. This string contains the name of the methods on the current call stack, together with the file name and line number where the exception was thrown for each method. A StackTrace object is created automatically by the common language runtime (CLR) from the point of the `throw` statement, so that exceptions must be thrown from the point where the stack trace should begin.

All exceptions contain a property named Message. This string should be set to explain the reason for the exception. Information that is sensitive to security shouldn't be put in the message text. In addition to Message, ArgumentException contains a property named ParamName that should be set to the name of the argument that caused the exception to be thrown. In a property setter, ParamName should be set to `value`.

Public and protected methods throw exceptions whenever they can't complete their intended functions. The exception class thrown is the most specific exception available that fits the error conditions. These exceptions should be documented as part of the

class functionality, and derived classes or updates to the original class should retain the same behavior for backward compatibility.

Things to avoid when throwing exceptions

The following list identifies practices to avoid when throwing exceptions:

- Don't use exceptions to change the flow of a program as part of ordinary execution. Use exceptions to report and handle error conditions.
- Exceptions shouldn't be returned as a return value or parameter instead of being thrown.
- Don't throw System.Exception, System.SystemException, System.NullReferenceException, or System.IndexOutOfRangeException intentionally from your own source code.
- Don't create exceptions that can be thrown in debug mode but not release mode. To identify run-time errors during the development phase, use Debug Assert instead.

Exceptions in task-returning methods

Methods declared with the `async` modifier have some special considerations when it comes to exceptions. Exceptions thrown in an `async` method are stored in the returned task and don't emerge until, for example, the task is awaited. For more information about stored exceptions, see Asynchronous exceptions.

We recommend that you validate arguments and throw any corresponding exceptions, such as ArgumentException and ArgumentNullException, before entering the asynchronous parts of your methods. That is, these validation exceptions should emerge synchronously before the work starts. The following code snippet shows an example where, if the exceptions are thrown, the ArgumentException exceptions would emerge synchronously, whereas the InvalidOperationException would be stored in the returned task.

C#

```
// Non-async, task-returning method.
// Within this method (but outside of the local function),
// any thrown exceptions emerge synchronously.
public static Task<Toast> ToastBreadAsync(int slices, int toastTime)
{
    if (slices is < 1 or > 4)
    {
        throw new ArgumentException(
```

```
            "You must specify between 1 and 4 slices of bread.",
            nameof(slices));
    }

    if (toastTime < 1)
    {
        throw new ArgumentException(
            "Toast time is too short.", nameof(toastTime));
    }

    return ToastBreadAsyncCore(slices, toastTime);

    // Local async function.
    // Within this function, any thrown exceptions are stored in the task.
    static async Task<Toast> ToastBreadAsyncCore(int slices, int time)
    {
        for (int slice = 0; slice < slices; slice++)
        {
            Console.WriteLine("Putting a slice of bread in the toaster");
        }
        // Start toasting.
        await Task.Delay(time);

        if (time > 2_000)
        {
            throw new InvalidOperationException("The toaster is on fire!");
        }

        Console.WriteLine("Toast is ready!");

        return new Toast();
    }
}
```

Define exception classes

Programs can throw a predefined exception class in the System namespace (except where previously noted), or create their own exception classes by deriving from Exception. The derived classes should define at least three constructors: one parameterless constructor, one that sets the message property, and one that sets both the Message and InnerException properties. For example:

C#

```
[Serializable]
public class InvalidDepartmentException : Exception
{
    public InvalidDepartmentException() : base() { }
    public InvalidDepartmentException(string message) : base(message) { }
    public InvalidDepartmentException(string message, Exception inner) :
```

```
    base(message, inner) { }
    }
```

Add new properties to the exception class when the data they provide is useful to resolving the exception. If new properties are added to the derived exception class, `ToString()` should be overridden to return the added information.

C# language specification

For more information, see Exceptions and The throw statement in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Exception Hierarchy

Compiler-generated exceptions

Article • 04/22/2023

Some exceptions are thrown automatically by the .NET runtime when basic operations fail. These exceptions and their error conditions are listed in the following table.

Expand table

Exception	Description
ArithmeticException	A base class for exceptions that occur during arithmetic operations, such as DivideByZeroException and OverflowException.
ArrayTypeMismatchException	Thrown when an array can't store a given element because the actual type of the element is incompatible with the actual type of the array.
DivideByZeroException	Thrown when an attempt is made to divide an integral value by zero.
IndexOutOfRangeException	Thrown when an attempt is made to index an array when the index is less than zero or outside the bounds of the array.
InvalidCastException	Thrown when an explicit conversion from a base type to an interface or to a derived type fails at run time.
NullReferenceException	Thrown when an attempt is made to reference an object whose value is null.
OutOfMemoryException	Thrown when an attempt to allocate memory using the new operator fails. This exception indicates that the memory available to the common language runtime has been exhausted.
OverflowException	Thrown when an arithmetic operation in a `checked` context overflows.
StackOverflowException	Thrown when the execution stack is exhausted by having too many pending method calls; usually indicates a very deep or infinite recursion.
TypeInitializationException	Thrown when a static constructor throws an exception and no compatible `catch` clause exists to catch it.

See also

- Exception-handling statements

C# identifier naming rules and conventions

10/10/2025

An **identifier** is the name you assign to a type (class, interface, struct, delegate, or enum), member, variable, or namespace.

This article covers the essential rules for valid C# identifiers and the naming conventions used to help you write consistent, professional code.

Naming rules

Valid identifiers must follow these rules. The C# compiler produces an error for any identifier that doesn't follow these rules:

- Identifiers must start with a letter or underscore (`_`).
- Identifiers can contain Unicode letter characters, decimal digit characters, Unicode connecting characters, Unicode combining characters, or Unicode formatting characters. For more information on Unicode categories, see the Unicode Category Database.

You can declare identifiers that match C# keywords by using the `@` prefix on the identifier. The `@` isn't part of the identifier name. For example, `@if` declares an identifier named `if`. These verbatim identifiers are primarily for interoperability with identifiers declared in other languages.

For a complete definition of valid identifiers, see the Identifiers article in the C# Language Specification.

> **Important**
>
> The C# language specification only allows letter (Lu, Ll, Lt, Lm, or Nl), digit (Nd), connecting (Pc), combining (Mn or Mc), and formatting (Cf) categories. Anything outside that is automatically replaced using `_`. This might impact certain Unicode characters.

Naming conventions

In addition to the rules, conventions for identifier names are used throughout the .NET APIs. These conventions provide consistency for names, but the compiler doesn't enforce them. You're free to use different conventions in your projects.

By convention, C# programs use `PascalCase` for type names, namespaces, and all public members. In addition, the `dotnet/docs` team uses the following conventions, adopted from the .NET Runtime team's coding style:

- Interface names start with a capital `I`.
- Attribute types end with the word `Attribute`.
- Enum types use a singular noun for nonflags, and a plural noun for flags.
- Identifiers shouldn't contain two consecutive underscore (`_`) characters. Those names are reserved for compiler-generated identifiers.
- Use meaningful and descriptive names for variables, methods, and classes.
- Prefer clarity over brevity.
- Use PascalCase for class names and method names.
- Use camelCase for method parameters and local variables.
- Use PascalCase for constant names, both fields and local constants.
- Private instance fields start with an underscore (`_`) and the remaining text is camelCased.
- Static fields start with `s_`. This convention isn't the default Visual Studio behavior, nor part of the Framework design guidelines, but is configurable in editorconfig.
- Avoid using abbreviations or acronyms in names, except for widely known and accepted abbreviations.
- Use meaningful and descriptive namespaces that follow the reverse domain name notation.
- Choose assembly names that represent the primary purpose of the assembly.
- Avoid using single-letter names, except for simple loop counters. Also, syntax examples that describe the syntax of C# constructs often use the following single-letter names that match the convention used in the C# language specification. Syntax examples are an exception to the rule.
  - Use `S` for structs, `C` for classes.
  - Use `M` for methods.
  - Use `v` for variables, `p` for parameters.
  - Use `r` for `ref` parameters.

> **Tip**
>
> You can enforce naming conventions that concern capitalization, prefixes, suffixes, and word separators by using code-style naming rules.

In the following examples, guidance pertaining to elements marked `public` is also applicable when working with `protected` and `protected internal` elements, all of which are intended to be visible to external callers.

Pascal case

Use pascal casing ("PascalCasing") when naming a `class`, `interface`, `struct`, or `delegate` type.

C#

```
public class DataService
{
}
```

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

C#

```
public struct ValueCoordinate
{
}
```

C#

```
public delegate void DelegateType(string message);
```

When naming an `interface`, use pascal casing in addition to prefixing the name with an `I`. This prefix clearly indicates to consumers that it's an `interface`.

C#

```
public interface IWorkerQueue
{
}
```

When naming `public` members of types, such as fields, properties, events, use pascal casing. Also, use pascal casing for all methods and local functions.

C#

```
public class ExampleEvents
{
    // A public field, these should be used sparingly
    public bool IsValid;

    // An init-only property
    public IWorkerQueue WorkerQueue { get; init; }

    // An event
    public event Action EventProcessing;

    // Method
    public void StartEventProcessing()
    {
        // Local function
        static int CountQueueItems() => WorkerQueue.Count;
        // ...
    }
}
```

When writing positional records, use pascal casing for parameters as they're the public properties of the record.

C#

```
public record PhysicalAddress(
    string Street,
    string City,
    string StateOrProvince,
    string ZipCode);
```

For more information on positional records, see Positional syntax for property definition.

Camel case

Use camel casing ("camelCasing") when naming `private` or `internal` fields and prefix them with `_`. Use camel casing when naming local variables, including instances of a delegate type.

C#

```
public class DataService
{
    private IWorkerQueue _workerQueue;
}
```

> **Tip**
>
> When editing C# code that follows these naming conventions in an IDE that supports statement completion, typing `_` will show all of the object-scoped members.

When working with `static` fields that are `private` or `internal`, use the `s_` prefix and for thread static use `t_`.

C#

```
public class DataService
{
    private static IWorkerQueue s_workerQueue;

    [ThreadStatic]
    private static TimeSpan t_timeSpan;
}
```

When writing method parameters, use camel casing.

C#

```
public T SomeMethod<T>(int someNumber, bool isValid)
{
}
```

Primary constructor parameters

How you name primary constructor parameters depends on the type being declared:

- For `class` and `struct` types: Use camel casing, consistent with other method parameters.

  C#

  ```
  public class DataService(IWorkerQueue workerQueue, ILogger logger)
  {
      public void ProcessData()
      {
  ```

```
        // Use the parameters directly
        logger.LogInformation("Processing data");
        workerQueue.Enqueue("data");
    }
}
```

C#

```
public struct Point(double x, double y)
{
    public double Distance => Math.Sqrt(x * x + y * y);
}
```

- For `record` types: Use Pascal casing, as the parameters become public properties.

C#

```
public record Person(string FirstName, string LastName);
public record Address(string Street, string City, string PostalCode);
```

For more information on primary constructors, see Primary constructors.

For more information on C# naming conventions, see the .NET Runtime team's coding style.

Type parameter naming guidelines

The following guidelines apply to type parameters on generic type parameters. Type parameters are the placeholders for arguments in a generic type or a generic method. You can read more about generic type parameters in the C# programming guide.

- **Do** name generic type parameters with descriptive names, unless a single letter name is completely self explanatory and a descriptive name wouldn't add value.

C#

```
public interface ISessionChannel<TSession> { /*...*/ }
public delegate TOutput Converter<TInput, TOutput>(TInput from);
public class List<T> { /*...*/ }
```

- **Consider** using `T` as the type parameter name for types with one single letter type parameter.

C#

```
public int IComparer<T>() => 0;
public delegate bool Predicate<T>(T item);
public struct Nullable<T> where T : struct { /*...*/ }
```

- **Do** prefix descriptive type parameter names with "T".

```
C#

public interface ISessionChannel<TSession>
{
    TSession Session { get; }
}
```

- **Consider** indicating constraints placed on a type parameter in the name of parameter. For example, a parameter constrained to `ISession` might be called `TSession`.

The code analysis rule CA1715 can be used to ensure that type parameters are named appropriately.

Extra naming conventions

- Examples that don't include using directives, use namespace qualifications. If you know that a namespace is imported by default in a project, you don't have to fully qualify the names from that namespace. Qualified names can be broken after a dot (.) if they're too long for a single line, as shown in the following example.

```
C#

var currentPerformanceCounterCategory = new System.Diagnostics.
    PerformanceCounterCategory();
```

- You don't have to change the names of objects that were created by using the Visual Studio designer tools to make them fit other guidelines.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Common C# code conventions

Article • 01/18/2025

Coding conventions are essential for maintaining code readability, consistency, and collaboration within a development team. Code that follows industry practices and established guidelines is easier to understand, maintain, and extend. Most projects enforce a consistent style through code conventions. The dotnet/docs and dotnet/samples projects are no exception. In this series of articles, you learn our coding conventions and the tools we use to enforce them. You can take our conventions as-is, or modify them to suit your team's needs.

We chose our conventions based on the following goals:

1. *Correctness*: Our samples are copied and pasted into your applications. We expect that, so we need to make code that's resilient and correct, even after multiple edits.
2. *Teaching*: The purpose of our samples is to teach all of .NET and C#. For that reason, we don't place restrictions on any language feature or API. Instead, those samples teach when a feature is a good choice.
3. *Consistency*: Readers expect a consistent experience across our content. All samples should conform to the same style.
4. *Adoption*: We aggressively update our samples to use new language features. That practice raises awareness of new features, and makes them more familiar to all C# developers.

> **Important**
>
> These guidelines are used by Microsoft to develop samples and documentation. They were adopted from the .NET Runtime, C# Coding Style and C# compiler (roslyn) guidelines. We chose those guidelines because of their adoption over several years of Open Source development. These guidelines help community members participate in the runtime and compiler projects. They're meant to be an example of common C# conventions, and not an authoritative list (see Framework Design Guidelines for detailed guidelines).
>
> The *teaching* and *adoption* goals are why the docs coding convention differs from the runtime and compiler conventions. Both the runtime and compiler have strict performance metrics for hot paths. Many other applications don't. Our *teaching* goal mandates that we don't prohibit any construct. Instead, samples show when constructs should be used. We update samples more aggressively than most

production applications do. Our *adoption* goal mandates that we show code you should write today, even when code written last year doesn't need changes.

This article explains our guidelines. The guidelines evolve over time, and you'll find samples that don't follow our guidelines. We welcome PRs that bring those samples into compliance, or issues that draw our attention to samples we should update. Our guidelines are Open Source and we welcome PRs and issues. However, if your submission would change these recommendations, open an issue for discussion first. You're welcome to use our guidelines, or adapt them to your needs.

Tools and analyzers

Tools can help your team enforce your conventions. You can enable code analysis to enforce the rules you prefer. You can also create an editorconfig so that Visual Studio automatically enforces your style guidelines. As a starting point, you can copy the dotnet/docs *.editorconfig* to use our style.

These tools make it easier for your team to adopt your preferred guidelines. Visual Studio applies the rules in all *.editorconfig* files in scope to format your code. You can use multiple configurations to enforce corporate-wide conventions, team conventions, and even granular project conventions.

Code analysis produces warnings and diagnostics when it detects rule violations. You configure the rules you want applied to your project. Then, each CI build notifies developers when they violate any of the rules.

Diagnostic IDs

- Choose appropriate diagnostic IDs when building your own analyzers

Language guidelines

The following sections describe practices that the .NET docs team follows to prepare code examples and samples. In general, follow these practices:

- Utilize modern language features and C# versions whenever possible.
- Avoid outdated language constructs.
- Only catch exceptions that can be properly handled; avoid catching general exceptions. For example, sample code shouldn't catch the System.Exception type without an exception filter.
- Use specific exception types to provide meaningful error messages.

- Use LINQ queries and methods for collection manipulation to improve code readability.
- Use asynchronous programming with async and await for I/O-bound operations.
- Be cautious of deadlocks and use Task.ConfigureAwait when appropriate.
- Use the language keywords for data types instead of the runtime types. For example, use `string` instead of System.String, or `int` instead of System.Int32. This recommendation includes using the types `nint` and `nuint`.
- Use `int` rather than unsigned types. The use of `int` is common throughout C#, and it's easier to interact with other libraries when you use `int`. Exceptions are for documentation specific to unsigned data types.
- Use `var` only when a reader can infer the type from the expression. Readers view our samples on the docs platform. They don't have hover or tool tips that display the type of variables.
- Write code with clarity and simplicity in mind.
- Avoid overly complex and convoluted code logic.

More specific guidelines follow.

String data

- Use string interpolation to concatenate short strings, as shown in the following code.

```
C#

string displayName = $"{nameList[n].LastName},
{nameList[n].FirstName}";
```

- To append strings in loops, especially when you're working with large amounts of text, use a System.Text.StringBuilder object.

```
C#

var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Prefer raw string literals to escape sequences or verbatim strings.

C#

```
var message = """
    This is a long message that spans across multiple lines.
    It uses raw string literals. This means we can
    also include characters like \n and \t without escaping them.
    """;
```

- Use the expression-based string interpolation rather than positional string interpolation.

C#

```
// Execute the queries.
Console.WriteLine("scoreQuery:");
foreach (var student in scoreQuery)
{
    Console.WriteLine($"{student.Last} Score: {student.score}");
}
```

Constructors and initialization

- Use Pascal case for primary constructor parameters on record types:

C#

```
public record Person(string FirstName, string LastName);
```

- Use camel case for primary constructor parameters on class and struct types.
- Use `required` properties instead of constructors to force initialization of property values:

C#

```
public class LabelledContainer<T>(string label)
{
    public string Label { get; } = label;
    public required T Contents
    {
        get;
        init;
    }
}
```

Arrays and collections

- Use collection expressions to initialize all collection types:

```
C#
```

```
string[] vowels = [ "a", "e", "i", "o", "u" ];
```

Delegates

- Use Func<> and Action<> instead of defining delegate types. In a class, define the delegate method.

```
C#
```

```
Action<string> actionExample1 = x => Console.WriteLine($"x is: {x}");

Action<string, string> actionExample2 = (x, y) =>
    Console.WriteLine($"x is: {x}, y is {y}");

Func<string, int> funcExample1 = x => Convert.ToInt32(x);

Func<int, int, int> funcExample2 = (x, y) => x + y;
```

- Call the method using the signature defined by the `Func<>` or `Action<>` delegate.

```
C#
```

```
actionExample1("string for x");

actionExample2("string for x", "string for y");

Console.WriteLine($"The value is {funcExample1("1")}");

Console.WriteLine($"The sum is {funcExample2(1, 2)}");
```

- If you create instances of a delegate type, use the concise syntax. In a class, define the delegate type and a method that has a matching signature.

  ```
  C#
  ```

  ```
  public delegate void Del(string message);

  public static void DelMethod(string str)
  {
  ```

```
    Console.WriteLine($"DelMethod argument: {str}");
}
```

- Create an instance of the delegate type and call it. The following declaration shows the condensed syntax.

```
C#

Del exampleDel2 = DelMethod;
exampleDel2("Hey");
```

- The following declaration uses the full syntax.

```
C#

Del exampleDel1 = new Del(DelMethod);
exampleDel1("Hey");
```

`try-catch` and `using` statements in exception handling

- Use a try-catch statement for most exception handling.

```
C#

static double ComputeDistance(double x1, double y1, double x2, double
y2)
{
    try
    {
        return Math.Sqrt((x1 - x2) * (x1 - x2) + (y1 - y2) * (y1 -
y2));
    }
    catch (System.ArithmeticException ex)
    {
        Console.WriteLine($"Arithmetic overflow or underflow: {ex}");
        throw;
    }
}
```

- Simplify your code by using the C# using statement. If you have a try-finally statement in which the only code in the `finally` block is a call to the Dispose method, use a `using` statement instead.

  In the following example, the `try-finally` statement only calls `Dispose` in the `finally` block.

C#

```
Font bodyStyle = new Font("Arial", 10.0f);
try
{
    byte charset = bodyStyle.GdiCharSet;
}
finally
{
    bodyStyle?.Dispose();
}
```

You can do the same thing with a `using` statement.

C#

```
using (Font arial = new Font("Arial", 10.0f))
{
    byte charset2 = arial.GdiCharSet;
}
```

Use the new using syntax that doesn't require braces:

C#

```
using Font normalStyle = new Font("Arial", 10.0f);
byte charset3 = normalStyle.GdiCharSet;
```

`&&` and `||` operators

- Use && instead of & and || instead of | when you perform comparisons, as shown in the following example.

C#

```
Console.Write("Enter a dividend: ");
int dividend = Convert.ToInt32(Console.ReadLine());

Console.Write("Enter a divisor: ");
int divisor = Convert.ToInt32(Console.ReadLine());

if ((divisor != 0) && (dividend / divisor) is var result)
{
    Console.WriteLine($"Quotient: {result}");
}
else
{
```

```
    Console.WriteLine("Attempted division by 0 ends up here.");
}
```

If the divisor is 0, the second clause in the `if` statement would cause a run-time error. But the && operator short-circuits when the first expression is false. That is, it doesn't evaluate the second expression. The & operator would evaluate both, resulting in a run-time error when `divisor` is 0.

`new` operator

- Use one of the concise forms of object instantiation when the variable type matches the object type, as shown in the following declarations. This form isn't valid when the variable is an interface type, or a base class of the runtime type.

  C#

  ```
  var firstExample = new ExampleClass();
  ```

  C#

  ```
  ExampleClass instance2 = new();
  ```

  The preceding declarations are equivalent to the following declaration.

  C#

  ```
  ExampleClass secondExample = new ExampleClass();
  ```

- Use object initializers to simplify object creation, as shown in the following example.

  C#

  ```
  var thirdExample = new ExampleClass { Name = "Desktop", ID = 37414,
      Location = "Redmond", Age = 2.3 };
  ```

  The following example sets the same properties as the preceding example but doesn't use initializers.

  C#

  ```
  var fourthExample = new ExampleClass();
  fourthExample.Name = "Desktop";
  ```

```
fourthExample.ID = 37414;
fourthExample.Location = "Redmond";
fourthExample.Age = 2.3;
```

Event handling

- Use a lambda expression to define an event handler that you don't need to remove later:

C#

```
public Form2()
{
    this.Click += (s, e) =>
        {
            MessageBox.Show(
                ((MouseEventArgs)e).Location.ToString());
        };
}
```

The lambda expression shortens the following traditional definition.

C#

```
public Form1()
{
    this.Click += new EventHandler(Form1_Click);
}

void Form1_Click(object? sender, EventArgs e)
{
    MessageBox.Show(((MouseEventArgs)e).Location.ToString());
}
```

Static members

Call static members by using the class name: *ClassName.StaticMember*. This practice makes code more readable by making static access clear. Don't qualify a static member defined in a base class with the name of a derived class. While that code compiles, the code readability is misleading, and the code might break in the future if you add a static member with the same name to the derived class.

LINQ queries

- Use meaningful names for query variables. The following example uses `seattleCustomers` for customers who are located in Seattle.

```
C#
```

```
var seattleCustomers = from customer in Customers
                       where customer.City == "Seattle"
                       select customer.Name;
```

- Use aliases to make sure that property names of anonymous types are correctly capitalized, using Pascal casing.

```
C#
```

```
var localDistributors =
    from customer in Customers
    join distributor in Distributors on customer.City equals
distributor.City
    select new { Customer = customer, Distributor = distributor };
```

- Rename properties when the property names in the result would be ambiguous. For example, if your query returns a customer name and a distributor name, instead of leaving them as a form of `Name` in the result, rename them to clarify `CustomerName` is the name of a customer, and `DistributorName` is the name of a distributor.

```
C#
```

```
var localDistributors2 =
    from customer in Customers
    join distributor in Distributors on customer.City equals
distributor.City
    select new { CustomerName = customer.Name, DistributorName =
distributor.Name };
```

- Use implicit typing in the declaration of query variables and range variables. This guidance on implicit typing in LINQ queries overrides the general rules for implicitly typed local variables. LINQ queries often use projections that create anonymous types. Other query expressions create results with nested generic types. Implicit typed variables are often more readable.

```
C#
```

```
var seattleCustomers = from customer in Customers
                       where customer.City == "Seattle"
```

```
                          select customer.Name;
```

- Align query clauses under the from clause, as shown in the previous examples.
- Use where clauses before other query clauses to ensure that later query clauses operate on the reduced, filtered set of data.

```
C#

var seattleCustomers2 = from customer in Customers
                        where customer.City == "Seattle"
                        orderby customer.Name
                        select customer;
```

- Access inner collections with multiple `from` clauses instead of a join clause. For example, a collection of `Student` objects might each contain a collection of test scores. When the following query is executed, it returns each score that is over 90, along with the family name of the student who received the score.

```
C#

var scoreQuery = from student in students
                 from score in student.Scores
                 where score > 90
                 select new { Last = student.LastName, score };
```

Implicitly typed local variables

- Use implicit typing for local variables when the type of the variable is obvious from the right side of the assignment.

```
C#

var message = "This is clearly a string.";
var currentTemperature = 27;
```

- Don't use var when the type isn't apparent from the right side of the assignment. Don't assume the type is clear from a method name. A variable type is considered clear if it's a `new` operator, an explicit cast, or assignment to a literal value.

```
C#

int numberOfIterations = Convert.ToInt32(Console.ReadLine());
```

```
int currentMaximum = ExampleClass.ResultSoFar();
```

- Don't use variable names to specify the type of the variable. It might not be correct. Instead, use the type to specify the type, and use the variable name to indicate the semantic information of the variable. The following example should use `string` for the type and something like `iterations` to indicate the meaning of the information read from the console.

```
C#

var inputInt = Console.ReadLine();
Console.WriteLine(inputInt);
```

- Avoid the use of `var` in place of dynamic. Use `dynamic` when you want run-time type inference. For more information, see Using type dynamic (C# Programming Guide).

- Use implicit typing for the loop variable in for loops.

  The following example uses implicit typing in a `for` statement.

```
C#

var phrase =
"lalalalalalalalalalalalalalalalalalalalalalalalalalalalalala";
var manyPhrases = new StringBuilder();
for (var i = 0; i < 10000; i++)
{
    manyPhrases.Append(phrase);
}
//Console.WriteLine("tra" + manyPhrases);
```

- Don't use implicit typing to determine the type of the loop variable in foreach loops. In most cases, the type of elements in the collection isn't immediately obvious. The collection's name shouldn't be solely relied upon for inferring the type of its elements.

  The following example uses explicit typing in a `foreach` statement.

```
C#

foreach (char ch in laugh)
{
    if (ch == 'h')
    {
        Console.Write("H");
```

```
    }
    else
    {
        Console.Write(ch);
    }
}
Console.WriteLine();
```

- use implicit type for the result sequences in LINQ queries. The section on LINQ explains that many LINQ queries result in anonymous types where implicit types must be used. Other queries result in nested generic types where `var` is more readable.

  > **Note**
  >
  > Be careful not to accidentally change a type of an element of the iterable collection. For example, it's easy to switch from System.Linq.IQueryable to System.Collections.IEnumerable in a `foreach` statement, which changes the execution of a query.

Some of our samples explain the *natural type* of an expression. Those samples must use `var` so that the compiler picks the natural type. Even though those examples are less obvious, the use of `var` is required for the sample. The text should explain the behavior.

File scoped namespace declarations

Most code files declare a single namespace. Therefore, our examples should use the file scoped namespace declarations:

```
C#

namespace MySampleCode;
```

Place the using directives outside the namespace declaration

When a `using` directive is outside a namespace declaration, that imported namespace is its fully qualified name. The fully qualified name is clearer. When the `using` directive is inside the namespace, it could be either relative to that namespace, or its fully qualified name.

```
```

C#

```
using Azure;

namespace CoolStuff.AwesomeFeature
{
    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Assuming there's a reference (direct, or indirect) to the WaitUntil class.

Now, let's change it slightly:

C#

```
namespace CoolStuff.AwesomeFeature
{
    using Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

And it compiles today. And tomorrow. But then sometime next week the preceding (untouched) code fails with two errors:

Console

```
- error CS0246: The type or namespace name 'WaitUntil' could not be found
(are you missing a using directive or an assembly reference?)
- error CS0103: The name 'WaitUntil' does not exist in the current context
```

One of the dependencies introduced this class in a namespace then ends with `.Azure`:

C#

```
namespace CoolStuff.Azure
{
    public class SecretsManagement
    {
        public string FetchFromKeyVault(string vaultId, string secretId) {
return null; }
    }
}
```

A `using` directive placed inside a namespace is context-sensitive and complicates name resolution. In this example, it's the first namespace that it finds.

- `CoolStuff.AwesomeFeature.Azure`
- `CoolStuff.Azure`
- `Azure`

Adding a new namespace that matches either `CoolStuff.Azure` or `CoolStuff.AwesomeFeature.Azure` would match before the global `Azure` namespace. You could resolve it by adding the `global::` modifier to the `using` declaration. However, it's easier to place `using` declarations outside the namespace instead.

C#

```
namespace CoolStuff.AwesomeFeature
{
    using global::Azure;

    public class Awesome
    {
        public void Stuff()
        {
            WaitUntil wait = WaitUntil.Completed;
            // ...
        }
    }
}
```

Style guidelines

In general, use the following format for code samples:

- Use four spaces for indentation. Don't use tabs.
- Align code consistently to improve readability.
- Limit lines to 65 characters to enhance code readability on docs, especially on mobile screens.

- Improve clarity and user experience by breaking long statements into multiple lines.
- Use the "Allman" style for braces: open and closing brace its own new line. Braces line up with current indentation level.
- Line breaks should occur before binary operators, if necessary.

Comment style

- Use single-line comments (`//`) for brief explanations.
- Avoid multi-line comments (`/* */`) for longer explanations.
  Comments in the code samples aren't localized. That means explanations embedded in the code aren't translated. Longer, explanatory text should be placed in the companion article, so that it can be localized.
- For describing methods, classes, fields, and all public members use XML comments.
- Place the comment on a separate line, not at the end of a line of code.
- Begin comment text with an uppercase letter.
- End comment text with a period.
- Insert one space between the comment delimiter (`//`) and the comment text, as shown in the following example.

```
C#

// The following declaration creates a query. It does not run
// the query.
```

Layout conventions

Good layout uses formatting to emphasize the structure of your code and to make the code easier to read. Microsoft examples and samples conform to the following conventions:

- Use the default Code Editor settings (smart indenting, four-character indents, tabs saved as spaces). For more information, see Options, Text Editor, C#, Formatting.
- Write only one statement per line.
- Write only one declaration per line.

- If continuation lines aren't indented automatically, indent them one tab stop (four spaces).
- Add at least one blank line between method definitions and property definitions.
- Use parentheses to make clauses in an expression apparent, as shown in the following code.

```
C#
```

```
if ((startX > endX) && (startX > previousX))
{
    // Take appropriate action.
}
```

Exceptions are when the sample explains operator or expression precedence.

Security

Follow the guidelines in Secure Coding Guidelines.

Tutorial: Build file-based C# programs

File-based apps are programs contained within a single `*.cs` file that you build and run without a corresponding project (`*.csproj`) file. File-based apps are ideal for learning C# because they have less complexity: The entire program is stored in a single file. File-based apps are also useful for building command line utilities. On Unix platforms, you can run file-based apps by using `#!` (shebang) directives. In this tutorial, you:

- ✓ Create a file-based app.
- ✓ Add Unix shebang (`#!`) support.
- ✓ Read command line arguments.
- ✓ Handle standard input.
- ✓ Write ASCII art output.
- ✓ Process command line arguments.
- ✓ Use parsed command line results.
- ✓ Test the final application.

You build a file-based app that writes text as ASCII art. The app is contained in a single file, uses NuGet packages, and implements core features.

Prerequisites

- The .NET 10 SDK. Download it from the .NET download site.
- Visual Studio Code. Download it from the Visual Studio Code homepage.
- (Optional) The C# DevKit extension for Visual Studio Code. Download it from the Visual Studio Code marketplace.

Create a file-based app

1. Open Visual Studio Code and create a new file named `AsciiArt.cs`. Enter the following text:

   C#

   ```
   Console.WriteLine("Hello, world!");
   ```

2. Save the file. Then, open the integrated terminal in Visual Studio Code and type:

   .NET CLI

   ```
   dotnet AsciiArt.cs
   ```

The first time you run this program, the `dotnet` host builds the executable from your source file, stores build artifacts in a temporary folder, and then runs the created executable. You can verify this experience by typing `dotnet AsciiArt.cs` again. This time, the `dotnet` host determines that the executable is current and runs the executable without building it again. You don't see any build output.

The preceding steps demonstrate that file-based apps aren't script files. They're C# source files that the `dotnet` host builds by using a generated project file in a temporary folder. One of the lines of output displayed when you build the program should look something like this (on Windows):

.NET CLI

```
AsciiArt succeeded (7.3s) → AppData\Local\Temp\dotnet\runfile\AsciiArt-
85c58ae0cd68371711f06f297fa0d7891d0de82afde04d8c64d5f910ddc04ddc\bin\debug\AsciiArt
.dll
```

On Unix platforms, the output folder is something similar to:

.NET CLI

```
AsciiArt succeeded (7.3s) → Library/Application Support/dotnet/runfile/AsciiArt-
85c58ae0cd68371711f06f297fa0d7891d0de82afde04d8c64d5f910ddc04ddc/bin/debug/AsciiArt
.dll
```

That output tells you where the temporary files and build outputs are placed. Throughout this tutorial, anytime you edit the source file, the `dotnet` host updates the executable before it runs.

File-based apps are regular C# programs. The only limitation is that you must write them in one source file. You can use top-level statements or a classic `Main` method as an entry point. You can declare any types: classes, interfaces, and structs. You can structure the algorithms in a file-based app the same as you would in any C# program. You can even declare multiple namespaces to organize your code. If you find a file-based app is growing too large for a single file, you can convert it to a project-based program and split the source into multiple files. File-based apps are a great prototyping tool. You can start experimenting with minimal overhead to prove concepts and build algorithms.

Unix shebang (`#!`) support

ⓘ Note

Support for `#!` directives applies on Unix platforms only. There's no similar directive for Windows to directly execute a C# program. On Windows, you must use `dotnet` on the command line.

On Unix, you can execute file-based apps directly using just the source file name. You need to make two changes:

1. Set *execute* permissions on the source file:

   Bash

   ```
   chmod +x AsciiArt.cs
   ```

2. Add a shebang (`#!`) directive as the first line of the `AsciiArt.cs` file:

   C#

   ```
   #!/usr/local/share/dotnet/dotnet
   ```

The location of `dotnet` can be different on different Unix installations. Use the command `which dotnet` to locate the `dotnet` host in your environment.

Alternatively, you can use `#!/usr/bin/env dotnet` to resolve the dotnet path from the PATH environment variable automatically:

C#

```
#!/usr/bin/env dotnet
```

After making these two changes, you can run the program directly:

Bash

```
./AsciiArt.cs
```

If you prefer, you can remove the extension so you can type `./AsciiArt` instead. You can add the `#!` to your source file even if you use Windows. Windows doesn't support `#!`, but the C# compiler allows that directive in file-based apps on all platforms.

Read command line arguments

Now, write all arguments on the command line to the output.

1. Replace the current contents of `AsciiArt.cs` with the following code:

   C#

   ```
   if (args.Length > 0)
   {
       string message = string.Join(' ', args);
       Console.WriteLine(message);
   }
   ```

2. You can run this version by typing the following command:

   .NET CLI

   ```
   dotnet AsciiArt.cs -- This is the command line.
   ```

   The `--` option indicates that all following command arguments should be passed to the AsciiArt program. The arguments `This is the command line.` are passed as an array of strings, where each string is one word: `This`, `is`, `the`, `command`, and `line.`.

This version demonstrates these new concepts:

- The predefined variable `args` passes the command line arguments to the program. The `args` variable is an array of strings: `string[]`. If the length of `args` is 0, no arguments were provided. Otherwise, each word on the argument list is stored in the corresponding entry in the array.
- The string.Join method joins multiple strings into a single string, with the specified separator. In this case, the separator is a single space.
- Console.WriteLine writes the string to the standard output console, followed by a new line.

Handle standard input

The preceding code handles command line arguments correctly. Now, add the code to handle reading input from standard input (`stdin`) instead of command line arguments.

1. Add the following `else` clause to the `if` statement you added in the preceding code:

   C#

   ```
   else
   {
       while (Console.ReadLine() is string line && line.Length > 0)
       {
   ```

```
        Console.WriteLine(line);
    }
}
```

The preceding code reads the console input until either a blank line or a `null` is read. (The Console.ReadLine method returns `null` if the input stream is closed by typing `ctrl+C`.)

2. Test reading standard input by creating a new text file in the same folder. Name the file `input.txt` and add the following lines:

   txt

   ```
   Hello from ...
   dotnet!

   You can create
   file-based apps
   in .NET 10 and
   C# 14

   Have fun writing
   useful utilities
   ```

   Keep the lines short so they format correctly when you add the feature to use ASCII art.

3. Run the program again.

   By using bash:

   Bash

   ```
   cat input.txt | dotnet AsciiArt.cs
   ```

   Or, by using PowerShell:

   PowerShell

   ```
   Get-Content input.txt | dotnet AsciiArt.cs
   ```

Now your program can accept either command line arguments or standard input.

Write ASCII art output

Next, add a package that supports ASCII art, Colorful.Console. To add a package to a file-based app, use the `#:package` directive.

1. Add the following directive after the `#!` directive in your `AsciiArt.cs` file:

   C#

   ```
   #:package Colorful.Console@1.2.15
   ```

   > **Important**
   >
   > The version `1.2.15` was the latest version of the `Colorful.Console` package when this tutorial was last updated. Check the package's NuGet page for the latest version to ensure you use a package version with the latest security fixes.

2. Change the lines that call `Console.WriteLine` to use the `Colorful.Console.WriteAscii` method instead:

   C#

   ```
   Colorful.Console.WriteAscii(line);
   ```

3. Run the program. You see ASCII art output instead of echoed text.

Process command options

Next, add command line parsing. The current version writes each word as a different line of output. The command line arguments you add support two features:

1. Quote multiple words that should be written on one line:

   .NET CLI

   ```
   AsciiArt.cs "This is line one" "This is another line" "This is the last line"
   ```

2. Add a `--delay` option to pause between each line:

   .NET CLI

   ```
   AsciiArt.cs --delay 1000
   ```

Users can use both arguments together.

Most command line applications need to parse command line arguments to handle options, commands, and user input effectively. The System.CommandLine library provides comprehensive capabilities to handle commands, subcommands, options, and arguments. You can concentrate on what your application does rather than the mechanics of parsing command line input.

The `System.CommandLine` library offers several key benefits:

- Automatic help text generation and validation.
- Support for POSIX and Windows command-line conventions.
- Built-in tab completion capabilities.
- Consistent parsing behavior across applications.

1. Add the `System.CommandLine` package. Add this directive after the existing package directive:

   C#

   ```
   #:package System.CommandLine@2.0.0
   ```

   > **ⓘ Important**
   >
   > The version `2.0.0` was the latest version when this tutorial was last updated. If there's a newer version available, use the latest version to ensure you have the latest security packages. Check the package's NuGet page for the latest version to ensure you use a package version with the latest security fixes.

2. Add the necessary using statements at the top of your file (after the `#!` and `#:package` directives):

   C#

   ```
   using System.CommandLine;
   using System.CommandLine.Parsing;
   ```

3. Define the delay option and messages argument. Add the following code to create the `CommandLine.Option` and `CommandLine.Argument` objects to represent the command line option and argument:

   C#

   ```
   Option<int> delayOption = new("--delay")
   {
   ```

```
    Description = "Delay between lines, specified as milliseconds.",
    DefaultValueFactory = parseResult => 100
};

Argument<string[]> messagesArgument = new("Messages")
{
    Description = "Text to render."
};
```

In command-line applications, options typically begin with `--` (double dash) and can accept arguments. The `--delay` option accepts an integer argument that specifies the delay in milliseconds. The `messagesArgument` defines how any remaining tokens after options are parsed as text. Each token becomes a separate string in the array, but text can be quoted to include multiple words in one token. For example, `"This is one message"` becomes a single token, while `This is four tokens` becomes four separate tokens.

The preceding code defines the argument type for the `--delay` option, and that the arguments are an array of `string` values. This application has only one command, so you use the *root command*.

4. Create a root command and configure it with the option and argument. Add the argument and option to the root command:

C#

```
RootCommand rootCommand = new("Ascii Art file-based app sample");

rootCommand.Options.Add(delayOption);
rootCommand.Arguments.Add(messagesArgument);
```

5. Add the code to parse the command line arguments and handle any errors. This code validates the command line arguments and stores parsed arguments in the System.CommandLine.ParseResult object:

C#

```
ParseResult result = rootCommand.Parse(args);
foreach (ParseError parseError in result.Errors)
{
    Console.Error.WriteLine(parseError.Message);
}
if (result.Errors.Count > 0)
{
    return 1;
}
```

The preceding code validates all command line arguments. If the validation fails, the app writes errors to the console and exits.

Use parsed command line results

Now, finish the app to use the parsed options and write the output. First, define a record to hold the parsed options. File-based apps can include type declarations, like records and classes. They must be after all top-level statements and local functions.

1. Add a record declaration to store the messages and the delay option value:

   C#

   ```
   public record AsciiMessageOptions(string[] Messages, int Delay);
   ```

2. Add the following local function before the record declaration. This method handles both command line arguments and standard input, and returns a new record instance:

   C#

   ```
   async Task<AsciiMessageOptions> ProcessParseResults(ParseResult result)
   {
       int delay = result.GetValue(delayOption);
       List<string> messages = [.. result.GetValue(messagesArgument) ??
   Array.Empty<string>()];

       if (messages.Count == 0)
       {
           while (Console.ReadLine() is string line && line.Length > 0)
           {
               Colorful.Console.WriteAscii(line);
               await Task.Delay(delay);
           }
       }
       return new([.. messages], delay);
   }
   ```

3. Create a local function to write the ASCII art with the specified delay. This function writes each message in the record with the specified delay between each message:

   C#

   ```
   async Task WriteAsciiArt(AsciiMessageOptions options)
   {
       foreach (string message in options.Messages)
       {
           Colorful.Console.WriteAscii(message);
   ```

```
        await Task.Delay(options.Delay);
    }
}
```

4. Replace the `if` clause you wrote earlier with the following code that processes the command line arguments and writes the output:

```C#
var parsedArgs = await ProcessParseResults(result);

await WriteAsciiArt(parsedArgs);
return 0;
```

You created a `record` type that provides structure to the parsed command line options and arguments. New local functions create an instance of the record, and use the record to write the ASCII art output.

Test the final application

Test the application by running several different commands. If you have trouble, here's the finished sample to compare with what you built:

```C#
#!/usr/bin/env dotnet

#:package Colorful.Console@1.2.15
#:package System.CommandLine@2.0.0

using System.CommandLine;
using System.CommandLine.Parsing;

Option<int> delayOption = new("--delay")
{
    Description = "Delay between lines, specified as milliseconds.",
    DefaultValueFactory = parseResult => 100
};

Argument<string[]> messagesArgument = new("Messages")
{
    Description = "Text to render."
};

RootCommand rootCommand = new("Ascii Art file-based app sample");

rootCommand.Options.Add(delayOption);
rootCommand.Arguments.Add(messagesArgument);
```

```
ParseResult result = rootCommand.Parse(args);
foreach (ParseError parseError in result.Errors)
{
    Console.Error.WriteLine(parseError.Message);
}
if (result.Errors.Count > 0)
{
    return 1;
}

var parsedArgs = await ProcessParseResults(result);

await WriteAsciiArt(parsedArgs);
return 0;

async Task<AsciiMessageOptions> ProcessParseResults(ParseResult result)
{
    int delay = result.GetValue(delayOption);
    List<string> messages = [.. result.GetValue(messagesArgument) ??
Array.Empty<string>()];

    if (messages.Count == 0)
    {
        while (Console.ReadLine() is string line && line.Length > 0)
        {
            // <WriteAscii>
            Colorful.Console.WriteAscii(line);
            // </WriteAscii>
            await Task.Delay(delay);
        }
    }
    return new([.. messages], delay);
}

async Task WriteAsciiArt(AsciiMessageOptions options)
{
    foreach (string message in options.Messages)
    {
        Colorful.Console.WriteAscii(message);
        await Task.Delay(options.Delay);
    }
}

public record AsciiMessageOptions(string[] Messages, int Delay);
```

In this tutorial, you learned to build a file-based app, where you build the program in a single C# file. These programs don't use a project file, and can use the `#!` directive on Unix systems. Learners can create these programs after trying our online tutorials and before building larger project-based apps. File-based apps are also a great platform for command line utilities.

Related content

- File-based apps reference
- Top level statement
- Preprocessor directives
- What's new in C# 14

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/25/2026

How to display command-line arguments

Article • 03/11/2022

Arguments provided to an executable on the command line are accessible in top-level statements or through an optional parameter to `Main`. The arguments are provided in the form of an array of strings. Each element of the array contains one argument. White-space between arguments is removed. For example, consider these command-line invocations of a fictitious executable:

Expand table

Input on command line	Array of strings passed to Main
executable.exe a b c	"a" "b" "c"
executable.exe one two	"one" "two"
executable.exe "one two" three	"one two" "three"

> **Note**
>
> When you are running an application in Visual Studio, you can specify command-line arguments in the Debug Page, Project Designer.

Example

This example displays the command-line arguments passed to a command-line application. The output shown is for the first entry in the table above.

C#

```
// The Length property provides the number of array elements.
Console.WriteLine($"parameter count = {args.Length}");
```

```
for (int i = 0; i < args.Length; i++)
{
    Console.WriteLine($"Arg[{i}] = [{args[i]}]");
}

/* Output (assumes 3 cmd line args):
    parameter count = 3
    Arg[0] = [a]
    Arg[1] = [b]
    Arg[2] = [c]
*/
```

See also

- System.CommandLine overview
- Tutorial: Get started with System.CommandLine

Tutorial: Build a command-line app with System.CommandLine

> **Tip**
>
> This article is part of the **Fundamentals** section, written for developers who know at least one programming language and are learning C#. If you're new to programming, start with Get started. If you need comprehensive library coverage, see the System.CommandLine library documentation.

The System.CommandLine library handles command-line parsing, help-text generation, and input validation so you can focus on your app's logic. In this tutorial, you build a task tracker CLI that demonstrates the core concepts: commands, subcommands, options, and arguments.

In this tutorial, you:

- ✓ Create a file-based app by using the `System.CommandLine` package.
- ✓ Define options and arguments with typed values.
- ✓ Build subcommands and attach options and arguments to them.
- ✓ Handle each subcommand with an action.
- ✓ Test the app with different command-line inputs.

Prerequisites

- Install the .NET 10 SDK or later.

Create the app

Start by creating a file-based C# program and adding the `System.CommandLine` package.

1. Create a file named `TaskCli.cs` with the following content:

   C#

   ```
   #!/usr/bin/env dotnet
   ```

   The `#!` (shebang) line lets you run the file directly on Unix systems. On Windows, run the file with `dotnet run TaskCli.cs`.

2. Add the `System.CommandLine` package directive and the required `using` statements:

C#

```
#:package System.CommandLine@2.0.0
```

C#

```
using System.CommandLine;
using System.CommandLine.Parsing;
using System.Text.Json;
```

> **ⓘ Important**
>
> Version `2.0.0` is the latest version at the time of writing. Check the package's NuGet page for the latest version to ensure you have the latest security fixes.

Understand the command structure

Before writing any parsing code, consider what the CLI looks like from the user's perspective. The task tracker supports four operations:

Console

```
dotnet TaskCli.cs -- add "Write documentation" --priority High --due 2026-04-01
dotnet TaskCli.cs -- list --all
dotnet TaskCli.cs -- complete 3
dotnet TaskCli.cs -- remove 3
dotnet TaskCli.cs -- --verbose list
```

> **ⓘ Note**
>
> The `--` after `TaskCli.cs` in the preceding examples tells dotnet run that all remaining arguments are passed to your app rather than interpreted by the `dotnet` CLI itself.

Each line uses several command-line concepts:

- **Subcommands** are verbs that tell the app what to do. The task tracker has four: `add`, `list`, `complete`, and `remove`. Each subcommand can define its own parameters.
- **Arguments** are positional values that follow a subcommand. In `add "Write documentation"`, the string `"Write documentation"` is an argument specifying the task description. In `complete 3`, the number `3` is an argument specifying the task ID.

- **Options** are named values prefixed with `--`. In `add --priority High --due 2026-04-01`, both `--priority` and `--due` are options with their own values. In `list --all`, the `--all` option is a Boolean flag that doesn't need a value.
- **Global options** apply to every subcommand. The `--verbose` option is defined on the root command with `Recursive = true`, so it works with any subcommand. In `--verbose list`, the verbose flag appears before the subcommand, but `list --verbose` works equally well.

In the sections that follow, you build these pieces from the bottom up. First, you define the individual options (like `--priority` and `--all`) and arguments (like the task description and ID). Next, you create the four subcommands and attach the relevant options and arguments to each one. Then you wire up an action for each subcommand. The action is the code that runs when the user invokes that command. Finally, you assemble the root command, parse the input, and invoke the matched action.

For a deeper look at command-line syntax concepts, see Command-line syntax overview.

Define options and arguments

Options represent named values that users specify with a `--` prefix. Arguments represent positional values. Both are strongly typed. `System.CommandLine` parses the input string into the type you specify.

The `System.CommandLine` library uses *generic types* to enforce type safety. When you write `Option<int>`, the `int` between the angle brackets is a *type argument*. It tells the library what type of value the option holds. The class itself declares a *type parameter* `T` (as in System.CommandLine.Option<T>), and you supply the concrete type when you create an instance. The library parses the user's input string and converts it to that type automatically. If the user provides `--delay abc` for an `Option<int>`, `System.CommandLine` reports a parse error instead of passing bad data to your code. You'll see this pattern with `Option<bool>`, `Option<Priority>`, `Option<DateOnly?>`, and System.CommandLine.Argument<T> in the steps that follow. For more information on generics, see Generics.

1. Define the options. Each `Option<T>` specifies the value type, the name, and a description. The `--priority` option uses an `enum` type, and `System.CommandLine` automatically validates the input against valid enum values:

   C#

   ```
   var verboseOption = new Option<bool>("--verbose")
   {
       Description = "Show detailed output",
   ```

```
    Recursive = true
};

var priorityOption = new Option<Priority>("--priority")
{
    Description = "Task priority level",
    DefaultValueFactory = _ => Priority.Medium
};

var dueOption = new Option<DateOnly?>("--due")
{
    Description = "Due date (uses current culture date format)"
};

var allOption = new Option<bool>("--all")
{
    Description = "Include completed tasks"
};
```

The `Recursive = true` setting on `--verbose` makes the option available on every subcommand. The `DefaultValueFactory` on `--priority` provides a default value so users can omit the option.

2. Define the arguments. Each `Argument<T>` specifies the value type and a name:

C#

```
var descriptionArgument = new Argument<string>("description")
{
    Description = "Task description"
};

var taskIdArgument = new Argument<int>("id")
{
    Description = "Task ID"
};
```

Build commands and subcommands

A System.CommandLine.Command represents an action that the user can invoke. Add the relevant options and arguments to each command so `System.CommandLine` knows which parameters belong where.

1. Create the four subcommands. Each command gets its own combination of options and arguments:

C#

```
var addCommand = new Command("add", "Add a new task")
{
    Arguments = { descriptionArgument },
    Options = { priorityOption, dueOption }
};
```

C#

```
var listCommand = new Command("list", "List all tasks")
{
    Options = { allOption }
};
```

C#

```
var completeCommand = new Command("complete", "Mark a task as complete")
{
    Arguments = { taskIdArgument }
};
```

C#

```
var removeCommand = new Command("remove", "Remove a task")
{
    Arguments = { taskIdArgument }
};
```

2. Assemble the root command. The System.CommandLine.RootCommand is the entry point for the CLI. Add the global `--verbose` option and all subcommands:

C#

```
var rootCommand = new RootCommand("A simple task tracker CLI")
{
    Options = { verboseOption },
    Subcommands = { addCommand, listCommand, completeCommand, removeCommand }
};
```

The autogenerated help text shows the root command description when the user runs `TaskCli --help`.

Handle commands with actions

Each subcommand needs an *action*. An action is a delegate that runs when the user invokes that command. A delegate is a type that represents a reference to a method. Here, you pass a

lambda expression (an inline anonymous function defined with `=>`) as the delegate. Call `SetAction` to assign each action. The delegate receives a ParseResult that provides access to parsed values through `GetValue`.

1. Set the action for the `add` command. This action introduces string interpolation (`$"..."` strings that embed expressions in braces), the conditional operator (`?:`), and a type test pattern (`due is DateOnly dueDate`) that checks whether a nullable value has a value and assigns it to a new variable in one step:

   C#

   ```
   addCommand.SetAction(parseResult =>
   {
       var description = parseResult.GetValue(descriptionArgument)!;
       var priority = parseResult.GetValue(priorityOption);
       var due = parseResult.GetValue(dueOption);
       var verbose = parseResult.GetValue(verboseOption);

       var tasks = LoadTasks();
       var id = tasks.Count > 0 ? tasks.Max(t => t.Id) + 1 : 1;
       var task = new TaskItem(id, description, priority, due, false);
       tasks.Add(task);
       SaveTasks(tasks);

       Console.WriteLine($"Added task {id}: {description}");
       if (verbose)
       {
           Console.WriteLine($"  Priority: {priority}");
           if (due is DateOnly dueDate)
           {
               Console.WriteLine($"  Due: {dueDate}");
           }
       }
   });
   ```

2. Set the action for the `list` command. LINQ (Language Integrated Query) gives you standard query operators for in-memory collections. In this action, Enumerable.Where filters the tasks to only the items that match a condition, and Enumerable.ToList materializes the filtered sequence into a list. The action then uses a foreach loop to iterate over the results, and the conditional operator to pick a status symbol:

   C#

   ```
   listCommand.SetAction(parseResult =>
   {
       var showAll = parseResult.GetValue(allOption);
       var verbose = parseResult.GetValue(verboseOption);
   ```

```
    var tasks = LoadTasks();
    var filtered = showAll ? tasks : tasks.Where(t => !t.IsComplete).ToList();

    if (filtered.Count == 0)
    {
        Console.WriteLine("No tasks found.");
        return;
    }

    foreach (var task in filtered)
    {
        var status = task.IsComplete ? "✓" : " ";
        Console.WriteLine($"  [{status}] {task.Id}: {task.Description}");
        if (verbose)
        {
            Console.WriteLine($"       Priority: {task.Priority}");
            if (task.Due is DateOnly dueDate)
            {
                Console.WriteLine($"       Due: {dueDate}");
            }
        }
    }
});
```

For more detail, see LINQ.

3. Set the action for the `complete` command. This action uses LINQ's Enumerable.FirstOrDefault to find:

    - A matching task.
    - An is null pattern to check whether the task exists.
    - A with expression to create a new record instance by copying the existing values first, and then applying the properties you set in the `with` initializer (here, `IsComplete = true`). Records are immutable by default, so this copy-and-update pattern is how you produce a modified value.

Because the action can fail (for example, the task ID doesn't exist), the action returns an integer error code that becomes the app's exit code:

C#

```
completeCommand.SetAction(parseResult =>
{
    var id = parseResult.GetValue(taskIdArgument);
    var verbose = parseResult.GetValue(verboseOption);

    var tasks = LoadTasks();
    var task = tasks.FirstOrDefault(t => t.Id == id);

    if (task is null)
```

```
    {
        Console.Error.WriteLine($"Task {id} not found.");
        return -1;
    }

    tasks[tasks.IndexOf(task)] = task with { IsComplete = true };
    SaveTasks(tasks);

    Console.WriteLine($"Completed task {id}: {task.Description}");
    if (verbose)
    {
        Console.WriteLine($"  Priority: {task.Priority}");
    }
    return 0;
});
```

4. Set the action for the `remove` command. This action follows the same look-up-and-validate pattern as `complete`. Use `FirstOrDefault` to find the task and `is null` to handle the missing case:

C#

```
removeCommand.SetAction(parseResult =>
{
    var id = parseResult.GetValue(taskIdArgument);
    var verbose = parseResult.GetValue(verboseOption);

    var tasks = LoadTasks();
    var task = tasks.FirstOrDefault(t => t.Id == id);

    if (task is null)
    {
        Console.Error.WriteLine($"Task {id} not found.");
        return -1;
    }

    tasks.Remove(task);
    SaveTasks(tasks);

    Console.WriteLine($"Removed task {id}: {task.Description}");
    if (verbose)
    {
        Console.WriteLine($"  Priority: {task.Priority}");
    }
    return 0;
});
```

5. Parse the command line and invoke the matched action:

C#

```
return rootCommand.Parse(args).Invoke();
```

`rootCommand.Parse(args)` parses the input into a ParseResult, and `.Invoke()` runs the action for the matched command. The return value is an exit code (0 for success).

Add supporting types and data helpers

The app needs a few supporting pieces: local functions, an enum, a record, and a serialization context. The following sections introduce each concept, explain why you'd choose it, and show the code. The app uses System.Text.Json to store tasks as JSON (JavaScript Object Notation). File-based apps require type declarations to appear after all top-level statements and local functions.

1. Add the local functions that load and save tasks. A local function is a method declared inside another function, including inside other local functions. Local functions keep helper logic close to the code that calls it, which improves readability because a reader doesn't have to jump to a separate class or file to understand the flow. Here, `LoadTasks` and `SaveTasks` encapsulate the file I/O that multiple command actions share, so the app writes the load/save logic once and reuses it:

C#

```
static string GetTaskFilePath() => Path.Combine(
    Environment.GetFolderPath(Environment.SpecialFolder.LocalApplicationData),
    "taskcli-sample",
    "tasks.json");

static List<TaskItem> LoadTasks()
{
    var path = GetTaskFilePath();
    if (!File.Exists(path))
    {
        return [];
    }

    var json = File.ReadAllText(path);
    return JsonSerializer.Deserialize(json,
TaskJsonContext.Default.ListTaskItem) ?? [];
}

static void SaveTasks(List<TaskItem> tasks)
{
    var path = GetTaskFilePath();
    Directory.CreateDirectory(Path.GetDirectoryName(path)!);
    var json = JsonSerializer.Serialize(tasks,
TaskJsonContext.Default.ListTaskItem);
```

```
    File.WriteAllText(path, json);
}
```

2. Add the `Priority` enum and `TaskItem` record at the end of the file:

```C#
public enum Priority { Low, Medium, High }
```

An enum is a value type that defines a fixed set of named constants backed by an integral type. You could represent priority levels with plain integers (0, 1, 2), but an `enum` is a better choice for several reasons: the compiler restricts assignments to the defined names, so a typo like `Hihg` causes a compile-time error instead of a silent bug; the names `Low`, `Medium`, and `High` make code more readable; and `System.CommandLine` automatically validates user input against the enum members, so you get free input checking.

```C#
public record TaskItem(int Id, string Description, Priority Priority,
DateOnly? Due, bool IsComplete);
```

A record is a type that the compiler equips with value-based equality and nondestructive mutation. A `record` is the right fit for `TaskItem` because task data is plain state with no complex behavior—you compare tasks by their values, not by reference identity. The compiler generates `Equals`, `GetHashCode`, `ToString`, and `with` expression support from the constructor parameters, so you get correct equality checks, easy debugging output, and immutable updates without writing boilerplate. Because records are immutable by default, you use a `with` expression to produce a modified copy (as the `complete` action does) rather than mutating the original, which prevents accidental side effects when different actions read and write the same list.

3. Add the JSON serialization context for AOT-compatible serialization:

```C#
[System.Text.Json.Serialization.JsonSourceGenerationOptions(WriteIndented =
true)]
[System.Text.Json.Serialization.JsonSerializable(typeof(List<TaskItem>))]
internal partial class TaskJsonContext :
System.Text.Json.Serialization.JsonSerializerContext;
```

This class uses two attributes (metadata annotations placed in square brackets above a declaration). The [JsonSourceGenerationOptions(WriteIndented = true)] attribute tells the

source generator to emit indented JSON for readability. The [JsonSerializable(typeof(List<TaskItem>))] attribute tells the generator which type to create serialization code for. Together, these attributes enable source-generated serialization, which avoids run-time reflection and supports ahead-of-time (AOT) compilation.

Test the application

Run the app with different inputs to exercise each subcommand.

1. View the autogenerated help:

.NET CLI

```
dotnet run TaskCli.cs -- --help
```

Output

```
Description:
  A simple task tracker CLI

Usage:
  TaskCli [command] [options]

Options:
  --verbose       Show detailed output
  -?, -h, --help  Show help and usage information
  --version       Show version information

Commands:
  add <description>  Add a new task
  list               List all tasks
  complete <id>      Mark a task as complete
  remove <id>        Remove a task
```

2. Add tasks with different options:

.NET CLI

```
dotnet run TaskCli.cs -- add "Write documentation" --priority High --due 2026-
04-01
dotnet run TaskCli.cs -- add "Review pull request"
dotnet run TaskCli.cs -- add "Fix build errors"
```

Output

```
Added task 1: Write documentation
Added task 2: Review pull request
Added task 3: Fix build errors
```

3. List tasks and use `--verbose` to show extra detail:

.NET CLI

```
dotnet run TaskCli.cs -- --verbose list
```

Output

```
  [ ] 1: Write documentation
       Priority: High
       Due: 4/1/2026
  [ ] 2: Review pull request
       Priority: Medium
  [ ] 3: Fix build errors
       Priority: Medium
```

4. Complete a task, then verify the list filters completed tasks by default:

.NET CLI

```
dotnet run TaskCli.cs -- complete 2
dotnet run TaskCli.cs -- list
```

Output

```
Completed task 2: Review pull request
  [ ] 1: Write documentation
  [ ] 3: Fix build errors
```

5. Use `--all` to include completed tasks:

.NET CLI

```
dotnet run TaskCli.cs -- list --all
```

Output

```
  [ ] 1: Write documentation
  [✓] 2: Review pull request
  [ ] 3: Fix build errors
```

6. Remove a task:

.NET CLI

```
dotnet run TaskCli.cs -- remove 3
```

Output

```
Removed task 3: Fix build errors
```

Clean up resources

The task tracker stores data in a JSON file under your local application data folder. Delete the `taskcli-sample` folder to remove the sample data:

- **Windows**: Delete `%LOCALAPPDATA%\taskcli-sample`.
- **macOS**: Delete `~/Library/Application Support/taskcli-sample`.
- **Linux**: Delete `~/.local/share/taskcli-sample`.

Related content

- System.CommandLine overview
- Tutorial: Get started with System.CommandLine
- Command-line syntax overview
- System.CommandLine design guidance
- Tutorial: Build file-based apps
- How to display command-line arguments

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 03/24/2026

Explore object oriented programming with classes and objects

10/10/2025

In this tutorial, you build a console application and see the basic object-oriented features that are part of the C# language.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Installation instructions

On Windows, this WinGet configuration file to install all prerequisites. If you already have something installed, WinGet will skip that step.

1. Download the file and double-click to run it.
2. Read the license agreement, type `y`, and select `Enter` when prompted to accept.
3. If you get a flashing User Account Control (UAC) prompt in your Taskbar, allow the installation to continue.

On other platforms, you need to install each of these components separately.

1. Download the recommended installer from the .NET SDK download page and double-click to run it. The download page detects your platform and recommends the latest installer for your platform.
2. Download the latest installer from the Visual Studio Code home page and double click to run it. That page also detects your platform and the link should be correct for your system.
3. Click the "Install" button on the C# DevKit extension page. That opens Visual Studio code, and asks if you want to install or enable the extension. Select "install".

Create your application

Using a terminal window, create a directory named *Classes*. You build your application there. Change to that directory and type `dotnet new console` in the console window. This command creates your application. Open *Program.cs*. It should look like this:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

In this tutorial, you're going to create new types that represent a bank account. Typically developers define each class in a different text file. That makes it easier to manage as a program grows in size. Create a new file named *BankAccount.cs* in the *Classes* directory.

This file contains the definition of a ***bank account***. Object Oriented programming organizes code by creating types in the form of ***classes***. These classes contain the code that represents a specific entity. The `BankAccount` class represents a bank account. The code implements specific operations through methods and properties. In this tutorial, the bank account supports this behavior:

1. It has a 10-digit number that uniquely identifies the bank account.
2. It has a string that stores the name or names of the owners.
3. The balance can be retrieved.
4. It accepts deposits.
5. It accepts withdrawals.
6. The initial balance must be positive.
7. Withdrawals can't result in a negative balance.

Define the bank account type

You can start by creating the basics of a class that defines that behavior. Create a new file using the **File:New** command. Name it *BankAccount.cs*. Add the following code to your *BankAccount.cs* file:

C#

```
namespace Classes;

public class BankAccount
{
    public string Number { get; }
    public string Owner { get; set; }
    public decimal Balance { get; }

    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
    }

    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
```

```
    }
}
```

Before going on, let's take a look at what you built. The `namespace` declaration provides a way to logically organize your code. This tutorial is relatively small, so you put all the code in one namespace.

`public class BankAccount` defines the class, or type, you're creating. Everything inside the `{` and `}` that follows the class declaration defines the state and behavior of the class. There are five ***members*** of the `BankAccount` class. The first three are ***properties***. Properties are data elements and can have code that enforces validation or other rules. The last two are ***methods***. Methods are blocks of code that perform a single function. Reading the names of each of the members should provide enough information for you or another developer to understand what the class does.

Open a new account

The first feature to implement is to open a bank account. When a customer opens an account, they must supply an initial balance, and information about the owner or owners of that account.

Creating a new object of the `BankAccount` type requires defining a ***constructor*** that assigns those values. A ***constructor*** is a member that has the same name as the class. It's used to initialize objects of that class type. Add the following constructor to the `BankAccount` type. Place the following code above the declaration of `MakeDeposit`:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    this.Owner = name;
    this.Balance = initialBalance;
}
```

The preceding code identifies the properties of the object being constructed by including the `this` qualifier. That qualifier is usually optional and omitted. You could also write:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Owner = name;
```

```
        Balance = initialBalance;
    }
```

The `this` qualifier is only required when a local variable or parameter has the same name as that field or property. The `this` qualifier is omitted throughout the remainder of this article unless it's necessary.

Constructors are called when you create an object using new. Replace the line `Console.WriteLine("Hello World!");` in *Program.cs* with the following code (replace `<name>` with your name):

C#

```
using Classes;

var account = new BankAccount("<name>", 1000);
Console.WriteLine($"Account {account.Number} was created for {account.Owner} with
{account.Balance} initial balance.");
```

Let's run what you built so far. If you're using Visual Studio, Select **Start without debugging** from the **Debug** menu. If you're using a command line, type `dotnet run` in the directory where you created your project.

Did you notice that the account number is blank? It's time to fix that. The account number should be assigned when the object is constructed. But it shouldn't be the responsibility of the caller to create it. The `BankAccount` class code should know how to assign new account numbers. A simple way is to start with a 10-digit number. Increment it when each new account is created. Finally, store the current account number when an object is constructed.

Add a member declaration to the `BankAccount` class. Place the following line of code after the opening brace `{` at the beginning of the `BankAccount` class:

C#

```
private static int s_accountNumberSeed = 1234567890;
```

The `accountNumberSeed` is a data member. It's `private`, which means it can only be accessed by code inside the `BankAccount` class. It's a way of separating the public responsibilities (like having an account number) from the private implementation (how account numbers are generated). It's also `static`, which means all `BankAccount` objects share the same single instance of this variable. The value of a non-static variable is unique to each instance of the `BankAccount` object. The `accountNumberSeed` is a `private static` field and thus has the `s_` prefix as per C# naming conventions. The `s` denoting `static` and `_` denoting `private` field. To

initialize each account number, add the following two lines to the constructor. Place them after the line that says `this.Balance = initialBalance`:

```
C#

Number = s_accountNumberSeed.ToString();
s_accountNumberSeed++;
```

Type `dotnet run` to see the results.

Create deposits and withdrawals

Your bank account class needs to accept deposits and withdrawals to work correctly. Let's implement deposits and withdrawals by creating a journal of every transaction for the account. Tracking every transaction has a few advantages over simply updating the balance on each transaction. The history can be used to audit all transactions and manage daily balances. Computing the balance from the history of all transactions when needed ensures any errors in a single transaction that are fixed will be correctly reflected in the balance on the next computation.

Start by creating a new type that represents a transaction. The transaction is a `record` type that doesn't have any responsibilities. It needs a few properties. Create a new file named *Transaction.cs*. Add the following code to it:

```
C#

namespace Classes;

public record Transaction(decimal Amount, DateTime Date, string Notes);
```

Now, let's add a List<T> of `Transaction` objects to the `BankAccount` class. Add the following declaration after the constructor in your *BankAccount.cs* file:

```
C#

private List<Transaction> _allTransactions = new List<Transaction>();
```

Now, let's correctly compute the `Balance`. The current balance can be found by summing the values of all transactions. As the code is currently, you can only get the initial balance of the account, so you have to update the `Balance` property. Replace the line `public decimal Balance { get; }` in *BankAccount.cs* with the following code:

```
```

C#

```
public decimal Balance
{
    get
    {
        decimal balance = 0;
        foreach (var item in _allTransactions)
        {
            balance += item.Amount;
        }

        return balance;
    }
}
```

This example shows an important aspect of ***properties***. You're now computing the balance when another programmer asks for the value. Your computation enumerates all transactions, and provides the sum as the current balance.

Next, implement the `MakeDeposit` and `MakeWithdrawal` methods. These methods enforce the final two rules: the initial balance must be positive, and any withdrawal must not create a negative balance.

These rules introduce the concept of ***exceptions***. The standard way of indicating that a method can't complete its work successfully is to throw an exception. The type of exception and the message associated with it describe the error. Here, the `MakeDeposit` method throws an exception if the amount of the deposit isn't greater than 0. The `MakeWithdrawal` method throws an exception if the withdrawal amount isn't greater than 0, or if applying the withdrawal results in a negative balance. Add the following code after the declaration of the `_allTransactions` list:

C#

```
public void MakeDeposit(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit
must be positive");
    }
    var deposit = new Transaction(amount, date, note);
    _allTransactions.Add(deposit);
}

public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
```

```
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < 0)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    _allTransactions.Add(withdrawal);
}
```

The throw statement **throws** an exception. Execution of the current block ends, and control transfers to the first matching `catch` block found in the call stack. You add a `catch` block to test this code a little later on.

The constructor should get one change so that it adds an initial transaction, rather than updating the balance directly. Since you already wrote the `MakeDeposit` method, call it from your constructor. The finished constructor should look like this:

C#

```
public BankAccount(string name, decimal initialBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

DateTime.Now is a property that returns the current date and time. Test this code by adding a few deposits and withdrawals in your `Main` method, following the code that creates a new `BankAccount`:

C#

```
account.MakeWithdrawal(500, DateTime.Now, "Rent payment");
Console.WriteLine(account.Balance);
account.MakeDeposit(100, DateTime.Now, "Friend paid me back");
Console.WriteLine(account.Balance);
```

Next, test that you're catching error conditions by trying to create an account with a negative balance. Add the following code after the preceding code you just added:

C#

```
// Test that the initial balances must be positive.
BankAccount invalidAccount;
try
{
    invalidAccount = new BankAccount("invalid", -55);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine("Exception caught creating account with negative balance");
    Console.WriteLine(e.ToString());
    return;
}
```

You use the try-catch statement to mark a block of code that might throw exceptions and to catch those errors that you expect. You can use the same technique to test the code that throws an exception for a negative balance. Add the following code before the declaration of `invalidAccount` in your `Main` method:

C#

```
// Test for a negative balance.
try
{
    account.MakeWithdrawal(750, DateTime.Now, "Attempt to overdraw");
}
catch (InvalidOperationException e)
{
    Console.WriteLine("Exception caught trying to overdraw");
    Console.WriteLine(e.ToString());
}
```

Save the file and type `dotnet run` to try it.

Challenge - log all transactions

To finish this tutorial, you can write the `GetAccountHistory` method that creates a `string` for the transaction history. Add this method to the `BankAccount` type:

C#

```
public string GetAccountHistory()
{
    var report = new System.Text.StringBuilder();

    decimal balance = 0;
    report.AppendLine("Date\t\tAmount\tBalance\tNote");
    foreach (var item in _allTransactions)
```

```
    {
        balance += item.Amount;
        report.AppendLine($"
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
    }

    return report.ToString();
}
```

The history uses the StringBuilder class to format a string that contains one line for each transaction. You used string formatting earlier in these tutorials. One new character is `\t`. That inserts a tab to format the output.

Add this line to test it in *Program.cs*:

```
C#

Console.WriteLine(account.GetAccountHistory());
```

Run your program and check the results.

Next steps

If you got stuck, you can see the source for this tutorial in our GitHub repo.

You can continue with the object oriented programming tutorial.

You can learn more about these concepts in these articles:

- Selection statements
- Iteration statements

Object-Oriented programming (C#)

10/10/2025

C# is an object-oriented programming language. The four basic principles of object-oriented programming are:

- *Abstraction* Modeling the relevant attributes and interactions of entities as classes to define an abstract representation of a system.
- *Encapsulation* Hiding the internal state and functionality of an object and only allowing access through a public set of functions.
- *Inheritance* Ability to create new abstractions based on existing abstractions.
- *Polymorphism* Ability to implement inherited properties or methods in different ways across multiple abstractions.

In the preceding tutorial, introduction to classes you saw both *abstraction* and *encapsulation*. The `BankAccount` class provided an abstraction for the concept of a bank account. You could modify its implementation without affecting any of the code that used the `BankAccount` class. Both the `BankAccount` and `Transaction` classes provide encapsulation of the components needed to describe those concepts in code.

In this tutorial, you extend that application to make use of *inheritance* and *polymorphism* to add new features. You also add features to the `BankAccount` class, taking advantage of the *abstraction* and *encapsulation* techniques you learned in the preceding tutorial.

Create different types of accounts

After building this program, you get requests to add features to it. It works great in the situation where there's only one bank account type. Over time, needs change, and related account types are requested:

- An interest earning account that accrues interest at the end of each month.
- A line of credit that can have a negative balance, but when there's a balance, there's an interest charge each month.
- A prepaid gift card account that starts with a single deposit, and only can be paid off. It can be refilled once at the start of each month.

All of these different accounts are similar to `BankAccount` class defined in the earlier tutorial. You could copy that code, rename the classes, and make modifications. That technique would work in the short term, but it would be more work over time. Any changes would be copied across all the affected classes.

Instead, you can create new bank account types that inherit methods and data from the `BankAccount` class created in the preceding tutorial. These new classes can extend the `BankAccount` class with the specific behavior needed for each type:

C#

```
public class InterestEarningAccount : BankAccount
{
}

public class LineOfCreditAccount : BankAccount
{
}

public class GiftCardAccount : BankAccount
{
}
```

Each of these classes *inherits* the shared behavior from their shared *base class*, the `BankAccount` class. Write the implementations for new and different functionality in each of the *derived classes*. These derived classes already have all the behavior defined in the `BankAccount` class.

It's a good practice to create each new class in a different source file. In Visual Studio, you can right-click on the project, and select *add class* to add a new class in a new file. In Visual Studio Code, select *File* then *New* to create a new source file. In either tool, name the file to match the class: *InterestEarningAccount.cs*, *LineOfCreditAccount.cs*, and *GiftCardAccount.cs*.

When you create the classes as shown in the preceding sample, you find that none of your derived classes compile. A constructor is responsible for initializing an object. A derived class constructor must initialize the derived class, and provide instructions on how to initialize the base class object included in the derived class. The proper initialization normally happens without any extra code. The `BankAccount` class declares one public constructor with the following signature:

C#

```
public BankAccount(string name, decimal initialBalance)
```

The compiler doesn't generate a default constructor when you define a constructor yourself. That means each derived class must explicitly call this constructor. You declare a constructor that can pass arguments to the base class constructor. The following code shows the constructor for the `InterestEarningAccount`:

C#

```
public InterestEarningAccount(string name, decimal initialBalance) : base(name,
initialBalance)
{
}
```

The parameters to this new constructor match the parameter type and names of the base class constructor. You use the `: base()` syntax to indicate a call to a base class constructor. Some classes define multiple constructors, and this syntax enables you to pick which base class constructor you call. Once you update the constructors, you can develop the code for each of the derived classes. The requirements for the new classes can be stated as follows:

- An interest earning account:
  - Gets a credit of 2% of the month-ending-balance.
- A line of credit:
  - Can have a negative balance, but not be greater in absolute value than the credit limit.
  - Incurs an interest charge each month where the end of month balance isn't 0.
  - Incurs a fee on each withdrawal that goes over the credit limit.
- A gift card account:
  - Can be refilled with a specified amount once each month, on the last day of the month.

You can see that all three of these account types have an action that takes places at the end of each month. However, each account type does different tasks. You use *polymorphism* to implement this code. Create a single `virtual` method in the `BankAccount` class:

C#

```
public virtual void PerformMonthEndTransactions() { }
```

The preceding code shows how you use the `virtual` keyword to declare a method in the base class that a derived class can provide a different implementation for. A `virtual` method is a method where any derived class can choose to reimplement. When a derived class defines a new implementation, it's called *overriding the base class implementation*. The derived classes use the `override` keyword to define the new implementation. The `virtual` keyword specifies that derived classes can override the behavior. You can also declare `abstract` methods where derived classes must override the behavior. The base class doesn't provide an implementation for an `abstract` method. Next, you need to define the implementation for two of the new classes you created. Start with the `InterestEarningAccount`:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance > 500m)
    {
        decimal interest = Balance * 0.02m;
        MakeDeposit(interest, DateTime.Now, "apply monthly interest");
    }
}
```

Add the following code to the `LineOfCreditAccount`. The code negates the balance to compute a positive interest charge that is withdrawn from the account:

C#

```
public override void PerformMonthEndTransactions()
{
    if (Balance < 0)
    {
        // Negate the balance to get a positive interest charge:
        decimal interest = -Balance * 0.07m;
        MakeWithdrawal(interest, DateTime.Now, "Charge monthly interest");
    }
}
```

The `GiftCardAccount` class needs two changes to implement its month-end functionality. First, modify the constructor to include an optional amount to add each month:

C#

```
private readonly decimal _monthlyDeposit = 0m;

public GiftCardAccount(string name, decimal initialBalance, decimal monthlyDeposit
= 0) : base(name, initialBalance)
    => _monthlyDeposit = monthlyDeposit;
```

The constructor provides a default value for the `monthlyDeposit` value so callers can omit a `0` for no monthly deposit. Next, override the `PerformMonthEndTransactions` method to add the monthly deposit, if it was set to a nonzero value in the constructor:

C#

```
public override void PerformMonthEndTransactions()
{
    if (_monthlyDeposit != 0)
    {
        MakeDeposit(_monthlyDeposit, DateTime.Now, "Add monthly deposit");
```

```
    }
}
```

The override applies the monthly deposit set in the constructor. Add the following code to the `Main` method. This code tests these changes for the `GiftCardAccount` and the `InterestEarningAccount`:

C#

```
var giftCard = new GiftCardAccount("gift card", 100, 50);
giftCard.MakeWithdrawal(20, DateTime.Now, "get expensive coffee");
giftCard.MakeWithdrawal(50, DateTime.Now, "buy groceries");
giftCard.PerformMonthEndTransactions();
// can make additional deposits:
giftCard.MakeDeposit(27.50m, DateTime.Now, "add some additional spending money");
Console.WriteLine(giftCard.GetAccountHistory());

var savings = new InterestEarningAccount("savings account", 10000);
savings.MakeDeposit(750, DateTime.Now, "save some money");
savings.MakeDeposit(1250, DateTime.Now, "Add more savings");
savings.MakeWithdrawal(250, DateTime.Now, "Needed to pay monthly bills");
savings.PerformMonthEndTransactions();
Console.WriteLine(savings.GetAccountHistory());
```

Verify the results. Now, add a similar set of test code for the `LineOfCreditAccount`:

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Add the preceding code and run the program. The output should be something like the following error:

Console

```
Unhandled exception. System.ArgumentOutOfRangeException: Amount of deposit must be
positive (Parameter 'amount')
   at OOProgramming.BankAccount.MakeDeposit(Decimal amount, DateTime date, String
note) in BankAccount.cs:line 42
   at OOProgramming.BankAccount..ctor(String name, Decimal initialBalance) in
BankAccount.cs:line 31
   at OOProgramming.LineOfCreditAccount..ctor(String name, Decimal initialBalance)
```

```
in LineOfCreditAccount.cs:line 9
   at OOProgramming.Program.Main(String[] args) in Program.cs:line 29
```

> **Note**
>
> The actual output includes the full path to the folder with the project. The folder names were omitted for brevity. Also, depending on your code format, the line numbers might be slightly different.

This code fails because the `BankAccount` assumes that the initial balance must be greater than 0. Another assumption baked into the `BankAccount` class is that the balance can't go negative. Instead, any withdrawal that overdraws the account is rejected. Both of those assumptions need to change. The line of credit account starts at 0, and generally has a negative balance. Also, if a customer borrows too much money, they incur a fee. The transaction is accepted, it just costs more. The first rule can be implemented by adding an optional argument to the `BankAccount` constructor that specifies the minimum balance. The default is `0`. The second rule requires a mechanism that enables derived classes to modify the default algorithm. In a sense, the base class "asks" the derived type what should happen when there's an overdraft. The default behavior is to reject the transaction by throwing an exception.

Let's start by adding a second constructor that includes an optional `minimumBalance` parameter. This new constructor does all the actions done by the existing constructor. Also, it sets the minimum balance property. You could copy the body of the existing constructor, but that means two locations to change in the future. Instead, you can use *constructor chaining* to have one constructor call another. The following code shows the two constructors and the new field:

C#

```
private readonly decimal _minimumBalance;

public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
{
    Number = s_accountNumberSeed.ToString();
    s_accountNumberSeed++;

    Owner = name;
    _minimumBalance = minimumBalance;
    if (initialBalance > 0)
        MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
}
```

The preceding code shows two new techniques. First, the `minimumBalance` field is marked as `readonly`. That means the value can't be changed after the object is constructed. Once a `BankAccount` is created, the `minimumBalance` can't change. Second, the constructor that takes two parameters uses `: this(name, initialBalance, 0) { }` as its implementation. The `: this()` expression calls the other constructor, the one with three parameters. This technique allows you to have a single implementation for initializing an object even though client code can choose one of many constructors.

This implementation calls `MakeDeposit` only if the initial balance is greater than `0`. That preserves the rule that deposits must be positive, yet lets the credit account open with a `0` balance.

Now that the `BankAccount` class has a read-only field for the minimum balance, the final change is to change the hard code `0` to `minimumBalance` in the `MakeWithdrawal` method:

```
C#

if (Balance - amount < _minimumBalance)
```

After extending the `BankAccount` class, you can modify the `LineOfCreditAccount` constructor to call the new base constructor, as shown in the following code:

```
C#

public LineOfCreditAccount(string name, decimal initialBalance, decimal
creditLimit) : base(name, initialBalance, -creditLimit)
{
}
```

Notice that the `LineOfCreditAccount` constructor changes the sign of the `creditLimit` parameter so it matches the meaning of the `minimumBalance` parameter.

Different overdraft rules

The last feature to add enables the `LineOfCreditAccount` to charge a fee for going over the credit limit instead of refusing the transaction.

One technique is to define a virtual function where you implement the required behavior. The `BankAccount` class refactors the `MakeWithdrawal` method into two methods. The new method does the specified action when the withdrawal takes the balance below the minimum. The existing `MakeWithdrawal` method has the following code:

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    if (amount <= 0)
    {
        throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
    }
    if (Balance - amount < _minimumBalance)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    var withdrawal = new Transaction(-amount, date, note);
    _allTransactions.Add(withdrawal);
}
```

Replace it with the following code:

C#

```
public void MakeWithdrawal(decimal amount, DateTime date, string note)
{
    ArgumentOutOfRangeException.ThrowIfNegativeOrZero(amount);
    Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance - amount <
_minimumBalance);
    Transaction? withdrawal = new(-amount, date, note);
    _allTransactions.Add(withdrawal);
    if (overdraftTransaction != null)
        _allTransactions.Add(overdraftTransaction);
}

protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
{
    if (isOverdrawn)
    {
        throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
    }
    else
    {
        return default;
    }
}
```

The added method is `protected`, which means that it can be called only from derived classes. That declaration prevents other clients from calling the method. It's also `virtual` so that derived classes can change the behavior. The return type is a `Transaction?`. The `?` annotation

indicates that the method can return `null`. Add the following implementation in the `LineOfCreditAccount`. This code charges a fee when the withdrawal limit is exceeded:

C#

```
protected override Transaction? CheckWithdrawalLimit(bool isOverdrawn) =>
    isOverdrawn
    ? new Transaction(-20, DateTime.Now, "Apply overdraft fee")
    : default;
```

The override returns a fee transaction when the account is overdrawn. If the withdrawal doesn't go over the limit, the method returns a `null` transaction. That indicates there's no fee. Test these changes by adding the following code to your `Main` method in the `Program` class:

C#

```
var lineOfCredit = new LineOfCreditAccount("line of credit", 0, 2000);
// How much is too much to borrow?
lineOfCredit.MakeWithdrawal(1000m, DateTime.Now, "Take out monthly advance");
lineOfCredit.MakeDeposit(50m, DateTime.Now, "Pay back small amount");
lineOfCredit.MakeWithdrawal(5000m, DateTime.Now, "Emergency funds for repairs");
lineOfCredit.MakeDeposit(150m, DateTime.Now, "Partial restoration on repairs");
lineOfCredit.PerformMonthEndTransactions();
Console.WriteLine(lineOfCredit.GetAccountHistory());
```

Run the program, and check the results.

Summary

If you got stuck, you can see the source for this tutorial in our GitHub repo.

This tutorial demonstrated many of the techniques used in Object-Oriented programming:

- You used *Abstraction* when you defined classes for each of the different account types. Those classes described the behavior for that type of account.
- You used *Encapsulation* when you kept many details `private` in each class.
- You used *Inheritance* when you used the implementation already created in the `BankAccount` class to save code.
- You used *Polymorphism* when you created `virtual` methods that derived classes could override to create specific behavior for that account type.

Inheritance in C# and .NET

Article • 02/03/2023

This tutorial introduces you to inheritance in C#. Inheritance is a feature of object-oriented programming languages that allows you to define a base class that provides specific functionality (data and behavior) and to define derived classes that either inherit or override that functionality.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Installation instructions

On Windows, this WinGet configuration file to install all prerequisites. If you already have something installed, WinGet will skip that step.

1. Download the file and double-click to run it.
2. Read the license agreement, type `y`, and select `Enter` when prompted to accept.
3. If you get a flashing User Account Control (UAC) prompt in your Taskbar, allow the installation to continue.

On other platforms, you need to install each of these components separately.

1. Download the recommended installer from the .NET SDK download page and double-click to run it. The download page detects your platform and recommends the latest installer for your platform.
2. Download the latest installer from the Visual Studio Code home page and double click to run it. That page also detects your platform and the link should be correct for your system.
3. Click the "Install" button on the C# DevKit extension page. That opens Visual Studio code, and asks if you want to install or enable the extension. Select "install".

Running the examples

To create and run the examples in this tutorial, you use the dotnet utility from the command line. Follow these steps for each example:

1. Create a directory to store the example.
2. Enter the dotnet new console command at a command prompt to create a new .NET Core project.
3. Copy and paste the code from the example into your code editor.
4. Enter the dotnet restore command from the command line to load or restore the project's dependencies.

   You don't have to run dotnet restore because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

   The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control when the restore occurs.

   For information about how to manage NuGet feeds, see the dotnet restore documentation.
5. Enter the dotnet run command to compile and execute the example.

Background: What is inheritance?

Inheritance is one of the fundamental attributes of object-oriented programming. It allows you to define a child class that reuses (inherits), extends, or modifies the behavior of a parent class. The class whose members are inherited is called the *base class*. The class that inherits the members of the base class is called the *derived class*.

C# and .NET support *single inheritance* only. That is, a class can only inherit from a single class. However, inheritance is transitive, which allows you to define an inheritance hierarchy for a set of types. In other words, type `D` can inherit from type `C`, which inherits from type `B`, which inherits from the base class type `A`. Because inheritance is transitive, the members of type `A` are available to type `D`.

Not all members of a base class are inherited by derived classes. The following members are not inherited:

- Static constructors, which initialize the static data of a class.
- Instance constructors, which you call to create a new instance of the class. Each class must define its own constructors.

- Finalizers, which are called by the runtime's garbage collector to destroy instances of a class.

While all other members of a base class are inherited by derived classes, whether they are visible or not depends on their accessibility. A member's accessibility affects its visibility for derived classes as follows:

- Private members are visible only in derived classes that are nested in their base class. Otherwise, they are not visible in derived classes. In the following example, `A.B` is a nested class that derives from `A`, and `C` derives from `A`. The private `A._value` field is visible in A.B. However, if you remove the comments from the `C.GetValue` method and attempt to compile the example, it produces compiler error CS0122: "'A._value' is inaccessible due to its protection level."

  C#

```
public class A
{
    private int _value = 10;

    public class B : A
    {
        public int GetValue()
        {
            return _value;
        }
    }
}

public class C : A
{
    //    public int GetValue()
    //    {
    //        return _value;
    //    }
}

public class AccessExample
{
    public static void Main(string[] args)
    {
        var b = new A.B();
        Console.WriteLine(b.GetValue());
    }
}
// The example displays the following output:
//       10
```

- Protected members are visible only in derived classes.

- Internal members are visible only in derived classes that are located in the same assembly as the base class. They are not visible in derived classes located in a different assembly from the base class.

- Public members are visible in derived classes and are part of the derived class' public interface. Public inherited members can be called just as if they are defined in the derived class. In the following example, class `A` defines a method named `Method1`, and class `B` inherits from class `A`. The example then calls `Method1` as if it were an instance method on `B`.

  C#

  ```
  public class A
  {
      public void Method1()
      {
          // Method implementation.
      }
  }

  public class B : A
  { }

  public class Example
  {
      public static void Main()
      {
          B b = new ();
          b.Method1();
      }
  }
  ```

Derived classes can also *override* inherited members by providing an alternate implementation. In order to be able to override a member, the member in the base class must be marked with the virtual keyword. By default, base class members are not marked as `virtual` and cannot be overridden. Attempting to override a non-virtual member, as the following example does, generates compiler error CS0506: "<member> cannot override inherited member <member> because it is not marked virtual, abstract, or override."

C#

```
public class A
{
    public void Method1()
    {
        // Do something.
    }
```

```
}

public class B : A
{
    public override void Method1() // Generates CS0506.
    {
        // Do something else.
    }
}
```

In some cases, a derived class *must* override the base class implementation. Base class members marked with the abstract keyword require that derived classes override them. Attempting to compile the following example generates compiler error CS0534, "<class> does not implement inherited abstract member <member>", because class `B` provides no implementation for `A.Method1`.

C#

```
public abstract class A
{
    public abstract void Method1();
}

public class B : A // Generates CS0534.
{
    public void Method3()
    {
        // Do something.
    }
}
```

Inheritance applies only to classes and interfaces. Other type categories (structs, delegates, and enums) do not support inheritance. Because of these rules, attempting to compile code like the following example produces compiler error CS0527: "Type 'ValueType' in interface list is not an interface." The error message indicates that, although you can define the interfaces that a struct implements, inheritance is not supported.

C#

```
public struct ValueStructure : ValueType // Generates CS0527.
{
}
```

Implicit inheritance

Besides any types that they may inherit from through single inheritance, all types in the .NET type system implicitly inherit from Object or a type derived from it. The common functionality of Object is available to any type.

To see what implicit inheritance means, let's define a new class, `SimpleClass`, that is simply an empty class definition:

C#

```
public class SimpleClass
{ }
```

You can then use reflection (which lets you inspect a type's metadata to get information about that type) to get a list of the members that belong to the `SimpleClass` type. Although you haven't defined any members in your `SimpleClass` class, output from the example indicates that it actually has nine members. One of these members is a parameterless (or default) constructor that is automatically supplied for the `SimpleClass` type by the C# compiler. The remaining eight are members of Object, the type from which all classes and interfaces in the .NET type system ultimately implicitly inherit.

C#

```
using System.Reflection;

public class SimpleClassExample
{
    public static void Main()
    {
        Type t = typeof(SimpleClass);
        BindingFlags flags = BindingFlags.Instance | BindingFlags.Static |
BindingFlags.Public |
                             BindingFlags.NonPublic |
BindingFlags.FlattenHierarchy;
        MemberInfo[] members = t.GetMembers(flags);
        Console.WriteLine($"Type {t.Name} has {members.Length} members: ");
        foreach (MemberInfo member in members)
        {
            string access = "";
            string stat = "";
            var method = member as MethodBase;
            if (method != null)
            {
                if (method.IsPublic)
                    access = " Public";
                else if (method.IsPrivate)
                    access = " Private";
                else if (method.IsFamily)
                    access = " Protected";
                else if (method.IsAssembly)
```

```
                    access = " Internal";
                else if (method.IsFamilyOrAssembly)
                    access = " Protected Internal ";
                if (method.IsStatic)
                    stat = " Static";
            }
            string output = $"{member.Name} ({member.MemberType}): {access}
{stat}, Declared by {member.DeclaringType}";
            Console.WriteLine(output);
        }
    }
}
// The example displays the following output:
//  Type SimpleClass has 9 members:
//  ToString (Method):  Public, Declared by System.Object
//  Equals (Method):  Public, Declared by System.Object
//  Equals (Method):  Public Static, Declared by System.Object
//  ReferenceEquals (Method):  Public Static, Declared by System.Object
//  GetHashCode (Method):  Public, Declared by System.Object
//  GetType (Method):  Public, Declared by System.Object
//  Finalize (Method):  Internal, Declared by System.Object
//  MemberwiseClone (Method):  Internal, Declared by System.Object
//  .ctor (Constructor):  Public, Declared by SimpleClass
```

Implicit inheritance from the Object class makes these methods available to the `SimpleClass` class:

- The public `ToString` method, which converts a `SimpleClass` object to its string representation, returns the fully qualified type name. In this case, the `ToString` method returns the string "SimpleClass".
- Three methods that test for equality of two objects: the public instance `Equals(Object)` method, the public static `Equals(Object, Object)` method, and the public static `ReferenceEquals(Object, Object)` method. By default, these methods test for reference equality; that is, to be equal, two object variables must refer to the same object.
- The public `GetHashCode` method, which computes a value that allows an instance of the type to be used in hashed collections.
- The public `GetType` method, which returns a Type object that represents the `SimpleClass` type.
- The protected Finalize method, which is designed to release unmanaged resources before an object's memory is reclaimed by the garbage collector.
- The protected MemberwiseClone method, which creates a shallow clone of the current object.

Because of implicit inheritance, you can call any inherited member from a `SimpleClass` object just as if it was actually a member defined in the `SimpleClass` class. For instance, the following example calls the `SimpleClass.ToString` method, which `SimpleClass` inherits from Object.

C#

```
public class EmptyClass
{ }

public class ClassNameExample
{
    public static void Main()
    {
        EmptyClass sc = new();
        Console.WriteLine(sc.ToString());
    }
}
// The example displays the following output:
//        EmptyClass
```

The following table lists the categories of types that you can create in C# and the types from which they implicitly inherit. Each base type makes a different set of members available through inheritance to implicitly derived types.

Expand table

Type category	Implicitly inherits from
class	Object
struct	ValueType, Object
enum	Enum, ValueType, Object
delegate	MulticastDelegate, Delegate, Object

Inheritance and an "is a" relationship

Ordinarily, inheritance is used to express an "is a" relationship between a base class and one or more derived classes, where the derived classes are specialized versions of the base class; the derived class is a type of the base class. For example, the `Publication` class represents a publication of any kind, and the `Book` and `Magazine` classes represent specific types of publications.

> **Note**
>
> A class or struct can implement one or more interfaces. While interface implementation is often presented as a workaround for single inheritance or as a way of using inheritance with structs, it is intended to express a different relationship (a "can do" relationship) between an interface and its implementing type than inheritance. An interface defines a subset of functionality (such as the ability to test for equality, to compare or sort objects, or to support culture-sensitive parsing and formatting) that the interface makes available to its implementing types.

Note that "is a" also expresses the relationship between a type and a specific instantiation of that type. In the following example, `Automobile` is a class that has three unique read-only properties: `Make`, the manufacturer of the automobile; `Model`, the kind of automobile; and `Year`, its year of manufacture. Your `Automobile` class also has a constructor whose arguments are assigned to the property values, and it overrides the Object.ToString method to produce a string that uniquely identifies the `Automobile` instance rather than the `Automobile` class.

C#

```
public class Automobile
{
    public Automobile(string make, string model, int year)
    {
        if (make == null)
            throw new ArgumentNullException(nameof(make), "The make cannot
be null.");
        else if (string.IsNullOrWhiteSpace(make))
            throw new ArgumentException("make cannot be an empty string or
have space characters only.");
        Make = make;

        if (model == null)
            throw new ArgumentNullException(nameof(model), "The model cannot
be null.");
        else if (string.IsNullOrWhiteSpace(model))
            throw new ArgumentException("model cannot be an empty string or
have space characters only.");
        Model = model;

        if (year < 1857 || year > DateTime.Now.Year + 2)
            throw new ArgumentException("The year is out of range.");
        Year = year;
    }

    public string Make { get; }
```

```
    public string Model { get; }

    public int Year { get; }

    public override string ToString() => $"{Year} {Make} {Model}";
}
```

In this case, you shouldn't rely on inheritance to represent specific car makes and models. For example, you don't need to define a `Packard` type to represent automobiles manufactured by the Packard Motor Car Company. Instead, you can represent them by creating an `Automobile` object with the appropriate values passed to its class constructor, as the following example does.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        var packard = new Automobile("Packard", "Custom Eight", 1948);
        Console.WriteLine(packard);
    }
}
// The example displays the following output:
//       1948 Packard Custom Eight
```

An is-a relationship based on inheritance is best applied to a base class and to derived classes that add additional members to the base class or that require additional functionality not present in the base class.

Designing the base class and derived classes

Let's look at the process of designing a base class and its derived classes. In this section, you'll define a base class, `Publication`, which represents a publication of any kind, such as a book, a magazine, a newspaper, a journal, an article, etc. You'll also define a `Book` class that derives from `Publication`. You could easily extend the example to define other derived classes, such as `Magazine`, `Journal`, `Newspaper`, and `Article`.

The base Publication class

In designing your `Publication` class, you need to make several design decisions:

- What members to include in your base `Publication` class, and whether the `Publication` members provide method implementations or whether `Publication` is an abstract base class that serves as a template for its derived classes.

  In this case, the `Publication` class will provide method implementations. The Designing abstract base classes and their derived classes section contains an example that uses an abstract base class to define the methods that derived classes must override. Derived classes are free to provide any implementation that is suitable for the derived type.

  The ability to reuse code (that is, multiple derived classes share the declaration and implementation of base class methods and do not need to override them) is an advantage of non-abstract base classes. Therefore, you should add members to `Publication` if their code is likely to be shared by some or most specialized `Publication` types. If you fail to provide base class implementations efficiently, you'll end up having to provide largely identical member implementations in derived classes rather a single implementation in the base class. The need to maintain duplicated code in multiple locations is a potential source of bugs.

  Both to maximize code reuse and to create a logical and intuitive inheritance hierarchy, you want to be sure that you include in the `Publication` class only the data and functionality that is common to all or to most publications. Derived classes then implement members that are unique to the particular kinds of publication that they represent.

- How far to extend your class hierarchy. Do you want to develop a hierarchy of three or more classes, rather than simply a base class and one or more derived classes? For example, `Publication` could be a base class of `Periodical`, which in turn is a base class of `Magazine`, `Journal` and `Newspaper`.

  For your example, you'll use the small hierarchy of a `Publication` class and a single derived class, `Book`. You could easily extend the example to create a number of additional classes that derive from `Publication`, such as `Magazine` and `Article`.

- Whether it makes sense to instantiate the base class. If it does not, you should apply the abstract keyword to the class. Otherwise, your `Publication` class can be instantiated by calling its class constructor. If an attempt is made to instantiate a class marked with the `abstract` keyword by a direct call to its class constructor, the C# compiler generates error CS0144, "Cannot create an instance of the abstract class or interface." If an attempt is made to instantiate the class by using reflection, the reflection method throws a MemberAccessException.

By default, a base class can be instantiated by calling its class constructor. You do not have to explicitly define a class constructor. If one is not present in the base class' source code, the C# compiler automatically provides a default (parameterless) constructor.

For your example, you'll mark the `Publication` class as abstract so that it cannot be instantiated. An `abstract` class without any `abstract` methods indicates that this class represents an abstract concept that is shared among several concrete classes (like a `Book`, `Journal`).

- Whether derived classes must inherit the base class implementation of particular members, whether they have the option to override the base class implementation, or whether they must provide an implementation. You use the abstract keyword to force derived classes to provide an implementation. You use the virtual keyword to allow derived classes to override a base class method. By default, methods defined in the base class are *not* overridable.

  The `Publication` class does not have any `abstract` methods, but the class itself is `abstract`.

- Whether a derived class represents the final class in the inheritance hierarchy and cannot itself be used as a base class for additional derived classes. By default, any class can serve as a base class. You can apply the sealed keyword to indicate that a class cannot serve as a base class for any additional classes. Attempting to derive from a sealed class generated compiler error CS0509, "cannot derive from sealed type <typeName>."

  For your example, you'll mark your derived class as `sealed`.

The following example shows the source code for the `Publication` class, as well as a `PublicationType` enumeration that is returned by the `Publication.PublicationType` property. In addition to the members that it inherits from Object, the `Publication` class defines the following unique members and member overrides:

C#

```
public enum PublicationType { Misc, Book, Magazine, Article };

public abstract class Publication
{
    private bool _published = false;
    private DateTime _datePublished;
    private int _totalPages;
```

```
    public Publication(string title, string publisher, PublicationType type)
    {
        if (string.IsNullOrWhiteSpace(publisher))
            throw new ArgumentException("The publisher is required.");
        Publisher = publisher;

        if (string.IsNullOrWhiteSpace(title))
            throw new ArgumentException("The title is required.");
        Title = title;

        Type = type;
    }

    public string Publisher { get; }

    public string Title { get; }

    public PublicationType Type { get; }

    public string? CopyrightName { get; private set; }

    public int CopyrightDate { get; private set; }

    public int Pages
    {
        get { return _totalPages; }
        set
        {
            if (value <= 0)
                throw new ArgumentOutOfRangeException(nameof(value), "The
number of pages cannot be zero or negative.");
            _totalPages = value;
        }
    }

    public string GetPublicationDate()
    {
        if (!_published)
            return "NYP";
        else
            return _datePublished.ToString("d");
    }

    public void Publish(DateTime datePublished)
    {
        _published = true;
        _datePublished = datePublished;
    }

    public void Copyright(string copyrightName, int copyrightDate)
    {
        if (string.IsNullOrWhiteSpace(copyrightName))
            throw new ArgumentException("The name of the copyright holder is
required.");
        CopyrightName = copyrightName;
```

```
        int currentYear = DateTime.Now.Year;
        if (copyrightDate < currentYear - 10 || copyrightDate > currentYear
+ 2)
            throw new ArgumentOutOfRangeException($"The copyright year must
be between {currentYear - 10} and {currentYear + 1}");
        CopyrightDate = copyrightDate;
    }

    public override string ToString() => Title;
}
```

- A constructor

  Because the `Publication` class is `abstract`, it cannot be instantiated directly from code like the following example:

  C#

  ```
  var publication = new Publication("Tiddlywinks for Experts", "Fun and
  Games",
                                    PublicationType.Book);
  ```

  However, its instance constructor can be called directly from derived class constructors, as the source code for the `Book` class shows.

- Two publication-related properties

  `Title` is a read-only String property whose value is supplied by calling the `Publication` constructor.

  `Pages` is a read-write Int32 property that indicates how many total pages the publication has. The value is stored in a private field named `totalPages`. It must be a positive number or an ArgumentOutOfRangeException is thrown.

- Publisher-related members

  Two read-only properties, `Publisher` and `Type`. The values are originally supplied by the call to the `Publication` class constructor.

- Publishing-related members

  Two methods, `Publish` and `GetPublicationDate`, set and return the publication date. The `Publish` method sets a private `published` flag to `true` when it is called and assigns the date passed to it as an argument to the private `datePublished`

field. The `GetPublicationDate` method returns the string "NYP" if the `published` flag is `false`, and the value of the `datePublished` field if it is `true`.

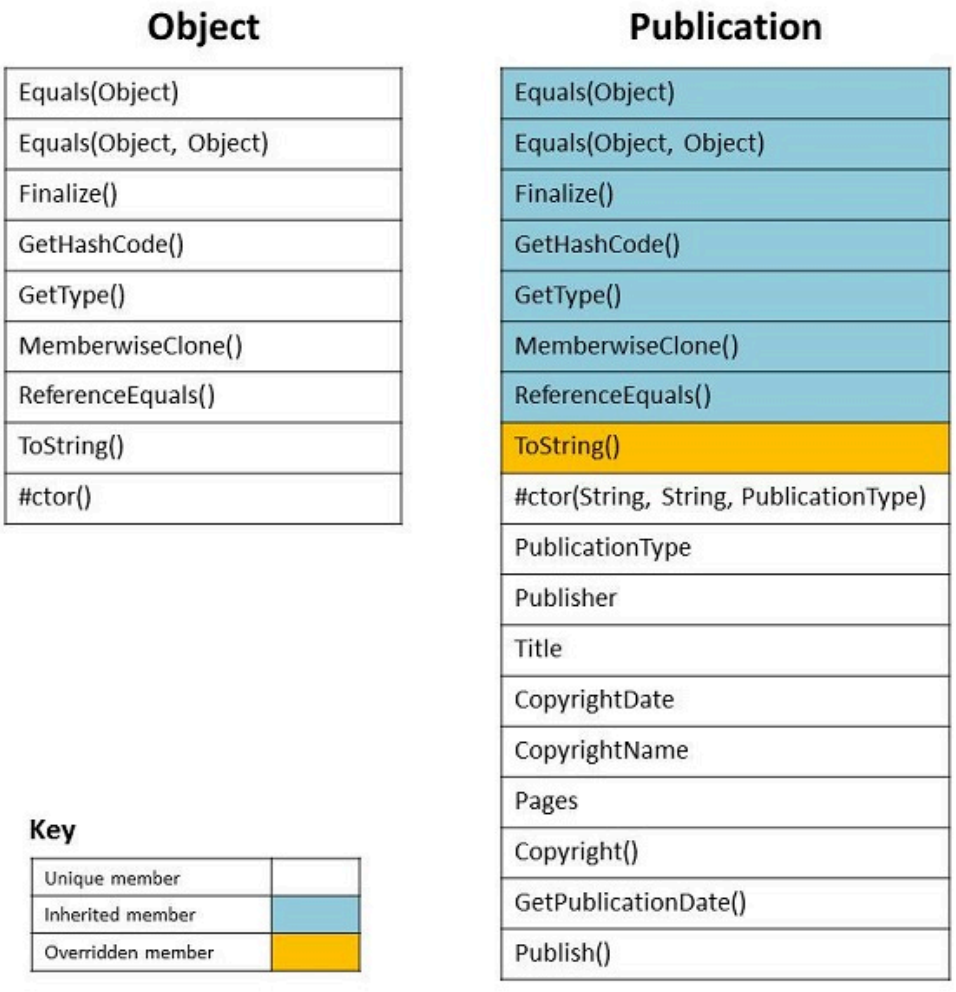
- Copyright-related members

  The `Copyright` method takes the name of the copyright holder and the year of the copyright as arguments and assigns them to the `CopyrightName` and `CopyrightDate` properties.

- An override of the `ToString` method

  If a type does not override the Object.ToString method, it returns the fully qualified name of the type, which is of little use in differentiating one instance from another. The `Publication` class overrides Object.ToString to return the value of the `Title` property.

The following figure illustrates the relationship between your base `Publication` class and its implicitly inherited Object class.



The `Book` class

The `Book` class represents a book as a specialized type of publication. The following example shows the source code for the `Book` class.

C#

```
using System;

public sealed class Book : Publication
{
    public Book(string title, string author, string publisher) :
           this(title, string.Empty, author, publisher)
    { }

    public Book(string title, string isbn, string author, string publisher)
: base(title, publisher, PublicationType.Book)
    {
        // isbn argument must be a 10- or 13-character numeric string
without "-" characters.
        // We could also determine whether the ISBN is valid by comparing
its checksum digit
        // with a computed checksum.
        //
        if (!string.IsNullOrEmpty(isbn))
        {
            // Determine if ISBN length is correct.
            if (!(isbn.Length == 10 | isbn.Length == 13))
                throw new ArgumentException("The ISBN must be a 10- or 13-
character numeric string.");
            if (!ulong.TryParse(isbn, out _))
                throw new ArgumentException("The ISBN can consist of numeric
characters only.");
        }
        ISBN = isbn;

        Author = author;
    }

    public string ISBN { get; }

    public string Author { get; }

    public decimal Price { get; private set; }

    // A three-digit ISO currency symbol.
    public string? Currency { get; private set; }

    // Returns the old price, and sets a new price.
    public decimal SetPrice(decimal price, string currency)
    {
        if (price < 0)
            throw new ArgumentOutOfRangeException(nameof(price), "The price
cannot be negative.");
        decimal oldValue = Price;
```

```
        Price = price;

        if (currency.Length != 3)
            throw new ArgumentException("The ISO currency symbol is a 3-
character string.");
        Currency = currency;

        return oldValue;
    }

    public override bool Equals(object? obj)
    {
        if (obj is not Book book)
            return false;
        else
            return ISBN == book.ISBN;
    }

    public override int GetHashCode() => ISBN.GetHashCode();

    public override string ToString() => $"{(string.IsNullOrEmpty(Author) ?
"" : Author + ", ")}{Title}";
}
```

In addition to the members that it inherits from `Publication`, the `Book` class defines the following unique members and member overrides:

- Two constructors

  The two `Book` constructors share three common parameters. Two, *title* and *publisher*, correspond to parameters of the `Publication` constructor. The third is *author*, which is stored to a public immutable `Author` property. One constructor includes an *isbn* parameter, which is stored in the `ISBN` auto-property.

  The first constructor uses the this keyword to call the other constructor. Constructor chaining is a common pattern in defining constructors. Constructors with fewer parameters provide default values when calling the constructor with the greatest number of parameters.

  The second constructor uses the base keyword to pass the title and publisher name to the base class constructor. If you don't make an explicit call to a base class constructor in your source code, the C# compiler automatically supplies a call to the base class' default or parameterless constructor.

- A read-only `ISBN` property, which returns the `Book` object's International Standard Book Number, a unique 10- or 13-digit number. The ISBN is supplied as an

argument to one of the `Book` constructors. The ISBN is stored in a private backing field, which is auto-generated by the compiler.

- A read-only `Author` property. The author name is supplied as an argument to both `Book` constructors and is stored in the property.

- Two read-only price-related properties, `Price` and `Currency`. Their values are provided as arguments in a `SetPrice` method call. The `Currency` property is the three-digit ISO currency symbol (for example, USD for the U.S. dollar). ISO currency symbols can be retrieved from the ISOCurrencySymbol property. Both of these properties are externally read-only, but both can be set by code in the `Book` class.

- A `SetPrice` method, which sets the values of the `Price` and `Currency` properties. Those values are returned by those same properties.

- Overrides to the `ToString` method (inherited from `Publication`) and the Object.Equals(Object) and GetHashCode methods (inherited from Object).

  Unless it is overridden, the Object.Equals(Object) method tests for reference equality. That is, two object variables are considered to be equal if they refer to the same object. In the `Book` class, on the other hand, two `Book` objects should be equal if they have the same ISBN.

  When you override the Object.Equals(Object) method, you must also override the GetHashCode method, which returns a value that the runtime uses to store items in hashed collections for efficient retrieval. The hash code should return a value that's consistent with the test for equality. Since you've overridden Object.Equals(Object) to return `true` if the ISBN properties of two `Book` objects are equal, you return the hash code computed by calling the GetHashCode method of the string returned by the `ISBN` property.

The following figure illustrates the relationship between the `Book` class and `Publication`, its base class.



You can now instantiate a `Book` object, invoke both its unique and inherited members, and pass it as an argument to a method that expects a parameter of type `Publication` or of type `Book`, as the following example shows.

C#

```
public class ClassExample
{
    public static void Main()
    {
        var book = new Book("The Tempest", "0971655819", "Shakespeare,
William",
```

```
                                "Public Domain Press");
        ShowPublicationInfo(book);
        book.Publish(new DateTime(2016, 8, 18));
        ShowPublicationInfo(book);

        var book2 = new Book("The Tempest", "Classic Works Press",
"Shakespeare, William");
        Console.Write($"{book.Title} and {book2.Title} are the same
publication: " +
                $"{((Publication)book).Equals(book2)}");
    }

    public static void ShowPublicationInfo(Publication pub)
    {
        string pubDate = pub.GetPublicationDate();
        Console.WriteLine($"{pub.Title}, " +
                  $"{(pubDate == "NYP" ? "Not Yet Published" : "published on
" + pubDate):d} by {pub.Publisher}");
    }
}
// The example displays the following output:
//        The Tempest, Not Yet Published by Public Domain Press
//        The Tempest, published on 8/18/2016 by Public Domain Press
//        The Tempest and The Tempest are the same publication: False
```

Designing abstract base classes and their derived classes

In the previous example, you defined a base class that provided an implementation for a number of methods to allow derived classes to share code. In many cases, however, the base class is not expected to provide an implementation. Instead, the base class is an *abstract class* that declares *abstract methods*; it serves as a template that defines the members that each derived class must implement. Typically in an abstract base class, the implementation of each derived type is unique to that type. You marked the class with the abstract keyword because it made no sense to instantiate a `Publication` object, although the class did provide implementations of functionality common to publications.

For example, each closed two-dimensional geometric shape includes two properties: area, the inner extent of the shape; and perimeter, or the distance along the edges of the shape. The way in which these properties are calculated, however, depends completely on the specific shape. The formula for calculating the perimeter (or circumference) of a circle, for example, is different from that of a square. The `Shape` class is an `abstract` class with `abstract` methods. That indicates derived classes share the same functionality, but those derived classes implement that functionality differently.

The following example defines an abstract base class named `Shape` that defines two properties: `Area` and `Perimeter`. In addition to marking the class with the abstract keyword, each instance member is also marked with the abstract keyword. In this case, `Shape` also overrides the Object.ToString method to return the name of the type, rather than its fully qualified name. And it defines two static members, `GetArea` and `GetPerimeter`, that allow callers to easily retrieve the area and perimeter of an instance of any derived class. When you pass an instance of a derived class to either of these methods, the runtime calls the method override of the derived class.

C#

```
public abstract class Shape
{
    public abstract double Area { get; }

    public abstract double Perimeter { get; }

    public override string ToString() => GetType().Name;

    public static double GetArea(Shape shape) => shape.Area;

    public static double GetPerimeter(Shape shape) => shape.Perimeter;
}
```

You can then derive some classes from `Shape` that represent specific shapes. The following example defines three classes, `Square`, `Rectangle`, and `Circle`. Each uses a formula unique for that particular shape to compute the area and perimeter. Some of the derived classes also define properties, such as `Rectangle.Diagonal` and `Circle.Diameter`, that are unique to the shape that they represent.

C#

```
using System;

public class Square : Shape
{
    public Square(double length)
    {
        Side = length;
    }

    public double Side { get; }

    public override double Area => Math.Pow(Side, 2);

    public override double Perimeter => Side * 4;

    public double Diagonal => Math.Round(Math.Sqrt(2) * Side, 2);
```

```
}

public class Rectangle : Shape
{
    public Rectangle(double length, double width)
    {
        Length = length;
        Width = width;
    }

    public double Length { get; }

    public double Width { get; }

    public override double Area => Length * Width;

    public override double Perimeter => 2 * Length + 2 * Width;

    public bool IsSquare() => Length == Width;

    public double Diagonal => Math.Round(Math.Sqrt(Math.Pow(Length, 2) +
Math.Pow(Width, 2)), 2);
}

public class Circle : Shape
{
    public Circle(double radius)
    {
        Radius = radius;
    }

    public override double Area => Math.Round(Math.PI * Math.Pow(Radius, 2),
2);

    public override double Perimeter => Math.Round(Math.PI * 2 * Radius, 2);

    // Define a circumference, since it's the more familiar term.
    public double Circumference => Perimeter;

    public double Radius { get; }

    public double Diameter => Radius * 2;
}
```

The following example uses objects derived from `Shape`. It instantiates an array of objects derived from `Shape` and calls the static methods of the `Shape` class, which wraps return `Shape` property values. The runtime retrieves values from the overridden properties of the derived types. The example also casts each `Shape` object in the array to its derived type and, if the cast succeeds, retrieves properties of that particular subclass of `Shape`.

C#

```
using System;

public class Example
{
    public static void Main()
    {
        Shape[] shapes = { new Rectangle(10, 12), new Square(5),
                           new Circle(3) };
        foreach (Shape shape in shapes)
        {
            Console.WriteLine($"{shape}: area, {Shape.GetArea(shape)}; " +
                              $"perimeter, {Shape.GetPerimeter(shape)}");
            if (shape is Rectangle rect)
            {
                Console.WriteLine($"   Is Square: {rect.IsSquare()},
Diagonal: {rect.Diagonal}");
                continue;
            }
            if (shape is Square sq)
            {
                Console.WriteLine($"   Diagonal: {sq.Diagonal}");
                continue;
            }
        }
    }
}
// The example displays the following output:
//         Rectangle: area, 120; perimeter, 44
//            Is Square: False, Diagonal: 15.62
//         Square: area, 25; perimeter, 20
//            Diagonal: 7.07
//         Circle: area, 28.27; perimeter, 18.85
```

Tutorial: Create XML documentation

In this tutorial, you take an existing object-oriented sample (from the preceding tutorial) and enhance it with XML documentation comments. XML documentation comments provide helpful IntelliSense tooltips, and can participate in generated API reference docs. You learn which elements deserve comments, how to use core tags like `<summary>`, `<param>`, `<returns>`, `<value>`, `<remarks>`, `<example>`, `<seealso>`, `<exception>`, and `<inheritdoc>`, and how consistent, purposeful commenting improves maintainability, discoverability, and collaboration—without adding noise. By the end, you annotated the public surface of the sample, built the project to emit the XML documentation file, and seen how those comments flow directly into the developer experience and downstream documentation tooling.

In this tutorial, you:

- ✓ Enable XML documentation output in your C# project.
- ✓ Add and structure XML documentation comments to types and members.
- ✓ Build the project and inspect the generated XML documentation file.

Prerequisites

- The .NET SDK
- Either Visual Studio or Visual Studio Code with the C# Dev Kit.

Enable XML documentation

Load the project you built in the preceding object-oriented tutorial. If you prefer to start fresh, clone the sample from the `dotnet/docs` repository under the snippets/object-oriented-programming folder.

Next, enable XML documentation output so the compiler emits a `.xml` file alongside your assembly. Edit the project file and add (or confirm) the following property inside a `<PropertyGroup>` element:

XML

```xml
<GenerateDocumentationFile>True</GenerateDocumentationFile>
```

If you're using Visual Studio, you can enable this using the "build" property page.

Build the project. The compiler produces an XML file that aggregates all the `///` comments from publicly visible types and members. That file feeds IntelliSense tooltips, static analysis tools, and downstream documentation generation systems.

Build the project now. You see warnings for any public members that are missing `<summary>` comments. Treat those warnings as a to-do list that helps you provide complete, intentional documentation. Open the generated XML file (it's next to your build output) and inspect the initial structure. At first, the `<members>` section is empty because you didn't add comments yet:

XML

```xml
<?xml version="1.0"?>
<doc>
    <assembly>
        <name>oo-programming</name>
    </assembly>
    <members>
    </members>
</doc>
```

With the file in place, start adding targeted XML comments and immediately verify how each one appears in the generated output. Start with the `Transaction` record type:

C#

```csharp
namespace OOProgramming;

/// <summary>
/// Represents an immutable financial transaction with an amount, date, and
descriptive notes.
/// </summary>
/// <param name="Amount">The transaction amount. Positive values represent
credits/deposits, negative values represent debits/withdrawals.</param>
/// <param name="Date">The date and time when the transaction occurred.</param>
/// <param name="Notes">Descriptive notes or memo text associated with the
transaction.</param>
public record Transaction(decimal Amount, DateTime Date, string Notes);
```

Add documentation comments

You now cycle through the build warnings to add concise, useful documentation to the `BankAccount` type. Each warning pinpoints a public member that lacks a `<summary>` (or other required) element. Treat the warning list as a checklist. Avoid adding noise: focus on describing intent, invariants, and important usage constraints—skip restating obvious type names or parameter types.

1. Build the project again. In Visual Studio or Visual Studio Code, open the Error List / Problems panel and filter for documentation warnings (CS1591). At the command line, run a build and review the warnings emitted to the console.

2. Navigate to the first warning (the `BankAccount` class). On the line above the declaration, type `///`. The editor scaffolds a `<summary>` element. Replace the placeholder with a single, action-focused sentence. The sentence explains the role of the account in the domain. For example, it tracks transactions and enforces a minimum balance.
3. Add `<remarks>` only if you need to explain behavior. Examples include how minimum balance enforcement works or how account numbers are generated. Keep remarks short.
4. For each property (`Number`, `Owner`, `Balance`), type `///` and write a `<summary>` that states what the value represents—not how a trivial getter returns it. If a property calculates a value (like `Balance`), add a `<value>` element that clarifies the calculation.
5. For each constructor, add `<summary>` plus `<param>` elements describing the meaning of each argument, not just restating the parameter name. If one overload delegates to another, add a concise `<remarks>` element.
6. For methods that can throw, add `<exception>` tags for each intentional exception type. Describe the condition that triggers it. Don't document exceptions thrown by argument validation helpers unless they're part of the public contract.
7. For methods that return a value, add `<returns>` with a short description of what callers receive. Avoid repeating the method name or managed type.
8. Work with the `BankAccount` base class first.

Your version should look something like the following code:

C#

```
namespace OOProgramming;

/// <summary>
/// Represents a bank account with basic banking operations including deposits,
withdrawals, and transaction history.
/// Supports minimum balance constraints and provides extensible month-end
processing capabilities.
/// </summary>
public class BankAccount
{
    /// <summary>
    /// Gets the unique account number for this bank account.
    /// </summary>
    /// <value>A string representation of the account number, generated
sequentially.</value>
    public string Number { get; }

    /// <summary>
    /// Gets or sets the name of the account owner.
    /// </summary>
    /// <value>The full name of the person who owns this account.</value>
    public string Owner { get; set; }

    /// <summary>
```

```
    /// Gets the current balance of the account by calculating the sum of all
transactions.
    /// </summary>
    /// <value>The current account balance as a decimal value.</value>
    public decimal Balance => _allTransactions.Sum(i => i.Amount);

    private static int s_accountNumberSeed = 1234567890;

    private readonly decimal _minimumBalance;

    /// <summary>
    /// Initializes a new instance of the BankAccount class with the specified owner
name and initial balance.
    /// Uses a default minimum balance of 0.
    /// </summary>
    /// <param name="name">The name of the account owner.</param>
    /// <param name="initialBalance">The initial deposit amount for the account.
</param>
    /// <remarks>
    /// This constructor is a convenience overload that calls the main constructor
with a minimum balance of 0.
    /// If the initial balance is greater than 0, it will be recorded as the first
transaction with the note "Initial balance".
    /// The account number is automatically generated using a static seed value that
increments for each new account.
    /// </remarks>
    public BankAccount(string name, decimal initialBalance) : this(name,
initialBalance, 0) { }

    /// <summary>
    /// Initializes a new instance of the BankAccount class with the specified owner
name, initial balance, and minimum balance constraint.
    /// </summary>
    /// <param name="name">The name of the account owner.</param>
    /// <param name="initialBalance">The initial deposit amount for the account.
</param>
    /// <param name="minimumBalance">The minimum balance that must be maintained in
the account.</param>
    /// <remarks>
    /// This is the primary constructor that sets up all account properties. The
account number is generated automatically
    /// using a static seed value. If an initial balance is provided and is greater
than 0, it will be added as the first
    /// transaction. The minimum balance constraint will be enforced on all future
withdrawal operations through the
    /// <see cref="CheckWithdrawalLimit"/> method.
    /// </remarks>
    public BankAccount(string name, decimal initialBalance, decimal minimumBalance)
    {
        Number = s_accountNumberSeed.ToString();
        s_accountNumberSeed++;

        Owner = name;
        _minimumBalance = minimumBalance;
        if (initialBalance > 0)
```

```
            MakeDeposit(initialBalance, DateTime.Now, "Initial balance");
    }

    private readonly List<Transaction> _allTransactions = [];

    /// <summary>
    /// Makes a deposit to the account by adding a positive transaction.
    /// </summary>
    /// <param name="amount">The amount to deposit. Must be positive.</param>
    /// <param name="date">The date when the deposit is made.</param>
    /// <param name="note">A descriptive note about the deposit transaction.</param>
    /// <exception cref="ArgumentOutOfRangeException">Thrown when the deposit amount
is zero or negative.</exception>
    /// <remarks>
    /// This method creates a new <see cref="Transaction"/> object with the
specified amount, date, and note,
    /// then adds it to the internal transaction list. The transaction amount must
be positive - negative amounts
    /// are not allowed for deposits. The account balance is automatically updated
through the Balance property
    /// which calculates the sum of all transactions. There are no limits or
restrictions on deposit amounts.
    /// </remarks>
    public void MakeDeposit(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of deposit
must be positive");
        }
        var deposit = new Transaction(amount, date, note);
        _allTransactions.Add(deposit);
    }

    /// <summary>
    /// Makes a withdrawal from the account by adding a negative transaction.
    /// Checks withdrawal limits and minimum balance constraints before processing.
    /// </summary>
    /// <param name="amount">The amount to withdraw. Must be positive.</param>
    /// <param name="date">The date when the withdrawal is made.</param>
    /// <param name="note">A descriptive note about the withdrawal transaction.
</param>
    /// <exception cref="ArgumentOutOfRangeException">Thrown when the withdrawal
amount is zero or negative.</exception>
    /// <exception cref="InvalidOperationException">Thrown when the withdrawal would
cause the balance to fall below the minimum balance.</exception>
    /// <remarks>
    /// This method first validates that the withdrawal amount is positive, then
checks if the withdrawal would
    /// violate the minimum balance constraint by calling <see
cref="CheckWithdrawalLimit"/>. The withdrawal is
    /// recorded as a negative transaction amount. If the withdrawal limit check
returns an overdraft transaction
    /// (such as a fee), that transaction is also added to the account. The method
enforces business rules through
```

```
    /// the virtual CheckWithdrawalLimit method, allowing derived classes to
implement different withdrawal policies.
    /// </remarks>
    public void MakeWithdrawal(decimal amount, DateTime date, string note)
    {
        if (amount <= 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Amount of
withdrawal must be positive");
        }
        Transaction? overdraftTransaction = CheckWithdrawalLimit(Balance - amount <
_minimumBalance);
        Transaction? withdrawal = new(-amount, date, note);
        _allTransactions.Add(withdrawal);
        if (overdraftTransaction != null)
            _allTransactions.Add(overdraftTransaction);
    }

    /// <summary>
    /// Checks whether a withdrawal would violate the account's minimum balance
constraint.
    /// This method can be overridden in derived classes to implement different
withdrawal limit policies.
    /// </summary>
    /// <param name="isOverdrawn">True if the withdrawal would cause the balance to
fall below the minimum balance.</param>
    /// <returns>A Transaction object representing any overdraft fees or penalties,
or null if no additional charges apply.</returns>
    /// <exception cref="InvalidOperationException">Thrown when the withdrawal would
cause an overdraft and the account type doesn't allow it.</exception>
    protected virtual Transaction? CheckWithdrawalLimit(bool isOverdrawn)
    {
        if (isOverdrawn)
        {
            throw new InvalidOperationException("Not sufficient funds for this
withdrawal");
        }
        else
        {
            return default;
        }
    }

    /// <summary>
    /// Generates a detailed account history report showing all transactions with
running balance calculations.
    /// </summary>
    /// <returns>A formatted string containing the complete transaction history with
dates, amounts, running balances, and notes.</returns>
    /// <remarks>
    /// This method creates a formatted report that includes a header row followed
by all transactions in chronological order.
    /// Each row shows the transaction date (in short date format), the transaction
amount, the running balance after that
    /// transaction, and any notes associated with the transaction. The running
```

```
balance is calculated by iterating through
    /// all transactions and maintaining a cumulative total. The report uses tab
characters for column separation and
    /// is suitable for display in console applications or simple text outputs.
    /// </remarks>
    public string GetAccountHistory()
    {
        var report = new System.Text.StringBuilder();

        decimal balance = 0;
        report.AppendLine("Date\t\tAmount\tBalance\tNote");
        foreach (var item in _allTransactions)
        {
            balance += item.Amount;
            report.AppendLine($"
{item.Date.ToShortDateString()}\t{item.Amount}\t{balance}\t{item.Notes}");
        }

        return report.ToString();
    }

    /// <summary>
    /// Performs month-end processing for the account. This virtual method can be
overridden in derived classes
    /// to implement specific month-end behaviors such as interest calculations, fee
assessments, or statement generation.
    /// </summary>
    /// <remarks>
    /// The base implementation of this method does nothing, providing a safe
default for basic bank accounts.
    /// Derived classes such as savings accounts or checking accounts can override
this method to implement
    /// account-specific month-end processing. Examples include calculating and
applying interest payments,
    /// assessing monthly maintenance fees, generating account statements, or
performing regulatory compliance checks.
    /// This method is typically called by banking systems at the end of each month
as part of batch processing operations.
    /// </remarks>
    public virtual void PerformMonthEndTransactions() { }
}
```

When you're done, open the regenerated XML file and confirm that each member appears with your new elements. A trimmed portion might look like this:

XML

```
<member name="T:OOProgramming.BankAccount">
  <summary>Represents a bank account that records transactions and enforces an
optional minimum balance.</summary>
  <remarks>Account numbers are generated sequentially when each instance is
constructed.</remarks>
</member>
```

```
<member name="P:OOProgramming.BankAccount.Balance">
  <summary>Gets the current balance based on all recorded transactions.</summary>
  <value>The net sum of deposits and withdrawals.</value>
</member>
```

> **Tip**
>
> Keep summaries to a single sentence. If you need more than one, move secondary context into `<remarks>`.

Use `<inheritdoc/>` in derived classes

If you derive from `BankAccount` (for example, a `SavingsAccount` that applies interest), you can inherit base documentation instead of copying it. Add a self-closing `<inheritdoc/>` element inside the derived member's documentation block. You can still append more elements (such as extra `<remarks>` details) after `<inheritdoc/>` to document the specialized behavior:

C#

```
/// <inheritdoc/>
/// <remarks>
/// An interest-earning account is a specialized savings account that rewards
customers for maintaining higher balances.
/// Interest is only earned when the account balance exceeds $500, encouraging
customers to maintain substantial deposits.
/// The annual interest rate of 2% is applied monthly to qualifying balances,
providing a simple savings incentive.
/// This account type uses the standard minimum balance of $0 from the base <see
cref="BankAccount"/> class.
/// </remarks>
public class InterestEarningAccount : BankAccount
```

> **Note**
>
> `<inheritdoc/>` reduces duplication and helps maintain consistency when you update base type documentation later.

After you finish documenting the public surface, build one final time to confirm there are no remaining CS1591 warnings. Your project now produces useful IntelliSense and a structured XML file ready for publishing workflows.

You can see the full annotated sample in the source folder of the dotnet/docs repository on GitHub.

Build output from comments

You can explore more by trying any of these tools to create output from XML comments:

- DocFX: *DocFX* is an API documentation generator for .NET, which currently supports C#, Visual Basic, and F#. It also allows you to customize the generated reference documentation. DocFX builds a static HTML website from your source code and Markdown files. Also, DocFX provides you with the flexibility to customize the layout and style of your website through templates. You can also create custom templates.
- Sandcastle: The *Sandcastle tools* create help files for managed class libraries containing both conceptual and API reference pages. The Sandcastle tools are command-line based and have no GUI front-end, project management features, or automated build process. The Sandcastle Help File Builder provides standalone GUI and command-line-based tools to build a help file in an automated fashion. A Visual Studio integration package is also available for it so that help projects can be created and managed entirely from within Visual Studio.
- Doxygen: *Doxygen* generates an online documentation browser (in HTML) or an offline reference manual (in LaTeX) from a set of documented source files. There's also support for generating output in RTF (MS Word), PostScript, hyperlinked PDF, compressed HTML, DocBook, and Unix manual pages. You can configure Doxygen to extract the code structure from undocumented source files.

Related content

- XML Doc language reference
- Recommended XML tags
- XML Documentation examples

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 12/03/2025

How to safely cast by using pattern matching and the is and as operators

Article • 03/11/2022

Because objects are polymorphic, it's possible for a variable of a base class type to hold a derived type. To access the derived type's instance members, it's necessary to cast the value back to the derived type. However, a cast creates the risk of throwing an InvalidCastException. C# provides pattern matching statements that perform a cast conditionally only when it will succeed. C# also provides the is and as operators to test if a value is of a certain type.

The following example shows how to use the pattern matching `is` statement:

C#

```
var g = new Giraffe();
var a = new Animal();
FeedMammals(g);
FeedMammals(a);
// Output:
// Eating.
// Animal is not a Mammal

SuperNova sn = new SuperNova();
TestForMammals(g);
TestForMammals(sn);

static void FeedMammals(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
    else
    {
        // variable 'm' is not in scope here, and can't be used.
        Console.WriteLine($"{a.GetType().Name} is not a Mammal");
    }
}

static void TestForMammals(object o)
{
    // You also can use the as operator and test for null
    // before referencing the variable.
    var m = o as Mammal;
    if (m != null)
    {
        Console.WriteLine(m.ToString());
```

```
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}
// Output:
// I am an animal.
// SuperNova is not a Mammal

class Animal
{
    public void Eat() { Console.WriteLine("Eating."); }
    public override string ToString()
    {
        return "I am an animal.";
    }
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

The preceding sample demonstrates a few features of pattern matching syntax. The `if (a is Mammal m)` statement combines the test with an initialization assignment. The assignment occurs only when the test succeeds. The variable `m` is only in scope in the embedded `if` statement where it has been assigned. You can't access `m` later in the same method. The preceding example also shows how to use the as operator to convert an object to a specified type.

You can also use the same syntax for testing if a nullable value type has a value, as shown in the following example:

C#

```
int i = 5;
PatternMatchingNullable(i);

int? j = null;
PatternMatchingNullable(j);

double d = 9.78654;
PatternMatchingNullable(d);

PatternMatchingSwitch(i);
PatternMatchingSwitch(j);
PatternMatchingSwitch(d);

static void PatternMatchingNullable(ValueType? val)
{
    if (val is int j) // Nullable types are not allowed in patterns
```

```
    {
        Console.WriteLine(j);
    }
    else if (val is null) // If val is a nullable type with no value, this
expression is true
    {
        Console.WriteLine("val is a nullable type with the null value");
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}

static void PatternMatchingSwitch(ValueType? val)
{
    switch (val)
    {
        case int number:
            Console.WriteLine(number);
            break;
        case long number:
            Console.WriteLine(number);
            break;
        case decimal number:
            Console.WriteLine(number);
            break;
        case float number:
            Console.WriteLine(number);
            break;
        case double number:
            Console.WriteLine(number);
            break;
        case null:
            Console.WriteLine("val is a nullable type with the null value");
            break;
        default:
            Console.WriteLine("Could not convert " + val.ToString());
            break;
    }
}
```

The preceding sample demonstrates other features of pattern matching to use with conversions. You can test a variable for the null pattern by checking specifically for the `null` value. When the runtime value of the variable is `null`, an `is` statement checking for a type always returns `false`. The pattern matching `is` statement doesn't allow a nullable value type, such as `int?` or `Nullable<int>`, but you can test for any other value type. The `is` patterns from the preceding example aren't limited to the nullable value types. You can also use those patterns to test if a variable of a reference type has a value or it's `null`.

The preceding sample also shows how you use the type pattern in a `switch` statement where the variable may be one of many different types.

If you want to test if a variable is a given type, but not assign it to a new variable, you can use the `is` and `as` operators for reference types and nullable value types. The following code shows how to use the `is` and `as` statements that were part of the C# language before pattern matching was introduced to test if a variable is of a given type:

C#

```
// Use the is operator to verify the type.
// before performing a cast.
Giraffe g = new();
UseIsOperator(g);

// Use the as operator and test for null
// before referencing the variable.
UseAsOperator(g);

// Use pattern matching to test for null
// before referencing the variable
UsePatternMatchingIs(g);

// Use the as operator to test
// an incompatible type.
SuperNova sn = new();
UseAsOperator(sn);

// Use the as operator with a value type.
// Note the implicit conversion to int? in
// the method body.
int i = 5;
UseAsWithNullable(i);

double d = 9.78654;
UseAsWithNullable(d);

static void UseIsOperator(Animal a)
{
    if (a is Mammal)
    {
        Mammal m = (Mammal)a;
        m.Eat();
    }
}

static void UsePatternMatchingIs(Animal a)
{
    if (a is Mammal m)
    {
        m.Eat();
    }
```

```
}

static void UseAsOperator(object o)
{
    Mammal? m = o as Mammal;
    if (m is not null)
    {
        Console.WriteLine(m.ToString());
    }
    else
    {
        Console.WriteLine($"{o.GetType().Name} is not a Mammal");
    }
}

static void UseAsWithNullable(System.ValueType val)
{
    int? j = val as int?;
    if (j is not null)
    {
        Console.WriteLine(j);
    }
    else
    {
        Console.WriteLine("Could not convert " + val.ToString());
    }
}
class Animal
{
    public void Eat() => Console.WriteLine("Eating.");
    public override string ToString() => "I am an animal.";
}
class Mammal : Animal { }
class Giraffe : Mammal { }

class SuperNova { }
```

As you can see by comparing this code with the pattern matching code, the pattern matching syntax provides more robust features by combining the test and the assignment in a single statement. Use the pattern matching syntax whenever possible.

Tutorial: Use pattern matching to build type-driven and data-driven algorithms

Article • 03/19/2025

You can write functionality that behaves as though you extended types that may be in other libraries. Another use for patterns is to create functionality your application requires that isn't a fundamental feature of the type being extended.

In this tutorial, you'll learn how to:

- ✓ Recognize situations where pattern matching should be used.
- ✓ Use pattern matching expressions to implement behavior based on types and property values.
- ✓ Combine pattern matching with other techniques to create complete algorithms.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Installation instructions

On Windows, this WinGet configuration file to install all prerequisites. If you already have something installed, WinGet will skip that step.

1. Download the file and double-click to run it.
2. Read the license agreement, type `y`, and select `Enter` when prompted to accept.
3. If you get a flashing User Account Control (UAC) prompt in your Taskbar, allow the installation to continue.

On other platforms, you need to install each of these components separately.

1. Download the recommended installer from the .NET SDK download page and double-click to run it. The download page detects your platform and recommends the latest installer for your platform.
2. Download the latest installer from the Visual Studio Code home page and double click to run it. That page also detects your platform and the link should be correct for your system.

3. Click the "Install" button on the C# DevKit extension page. That opens Visual Studio code, and asks if you want to install or enable the extension. Select "install".

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Scenarios for pattern matching

Modern development often includes integrating data from multiple sources and presenting information and insights from that data in a single cohesive application. You and your team won't have control or access for all the types that represent the incoming data.

The classic object-oriented design would call for creating types in your application that represent each data type from those multiple data sources. Then, your application would work with those new types, build inheritance hierarchies, create virtual methods, and implement abstractions. Those techniques work, and sometimes they're the best tools. Other times you can write less code. You can write more clear code using techniques that separate the data from the operations that manipulate that data.

In this tutorial, you'll create and explore an application that takes incoming data from several external sources for a single scenario. You'll see how **pattern matching** provides an efficient way to consume and process that data in ways that weren't part of the original system.

Consider a major metropolitan area that is using tolls and peak time pricing to manage traffic. You write an application that calculates tolls for a vehicle based on its type. Later enhancements incorporate pricing based on the number of occupants in the vehicle. Further enhancements add pricing based on the time and the day of the week.

From that brief description, you may have quickly sketched out an object hierarchy to model this system. However, your data is coming from multiple sources like other vehicle registration management systems. These systems provide different classes to model that data and you don't have a single object model you can use. In this tutorial, you'll use these simplified classes to model for the vehicle data from these external systems, as shown in the following code:

C#

```
namespace ConsumerVehicleRegistration
{
    public class Car
    {
        public int Passengers { get; set; }
```

```
    }
}

namespace CommercialRegistration
{
    public class DeliveryTruck
    {
        public int GrossWeightClass { get; set; }
    }
}

namespace LiveryRegistration
{
    public class Taxi
    {
        public int Fares { get; set; }
    }

    public class Bus
    {
        public int Capacity { get; set; }
        public int Riders { get; set; }
    }
}
```

You can download the starter code from the dotnet/samples GitHub repository. You can see that the vehicle classes are from different systems, and are in different namespaces. No common base class, other than `System.Object` can be used.

Pattern matching designs

The scenario used in this tutorial highlights the kinds of problems that pattern matching is well suited to solve:

- The objects you need to work with aren't in an object hierarchy that matches your goals. You may be working with classes that are part of unrelated systems.
- The functionality you're adding isn't part of the core abstraction for these classes. The toll paid by a vehicle *changes* for different types of vehicles, but the toll isn't a core function of the vehicle.

When the *shape* of the data and the *operations* on that data aren't described together, the pattern matching features in C# make it easier to work with.

Implement the basic toll calculations

The most basic toll calculation relies only on the vehicle type:

- A `Car` is $2.00.
- A `Taxi` is $3.50.
- A `Bus` is $5.00.
- A `DeliveryTruck` is $10.00

Create a new `TollCalculator` class, and implement pattern matching on the vehicle type to get the toll amount. The following code shows the initial implementation of the `TollCalculator`.

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

namespace Calculators;

public class TollCalculator
{
    public decimal CalculateToll(object vehicle) =>
        vehicle switch
    {
        Car c           => 2.00m,
        Taxi t          => 3.50m,
        Bus b           => 5.00m,
        DeliveryTruck t => 10.00m,
        { }             => throw new ArgumentException(message: "Not a known
vehicle type", paramName: nameof(vehicle)),
        null            => throw new ArgumentNullException(nameof(vehicle))
    };
}
```

The preceding code uses a switch expression (not the same as a switch statement) that tests the declaration pattern. A **switch expression** begins with the variable, `vehicle` in the preceding code, followed by the `switch` keyword. Next comes all the **switch arms** inside curly braces. The `switch` expression makes other refinements to the syntax that surrounds the `switch` statement. The `case` keyword is omitted, and the result of each arm is an expression. The last two arms show a new language feature. The `{ }` case matches any non-null object that didn't match an earlier arm. This arm catches any incorrect types passed to this method. The `{ }` case must follow the cases for each vehicle type. If the order were reversed, the `{ }` case would take precedence. Finally, the `null` constant pattern detects when `null` is passed to this method. The `null` pattern can be last because the other patterns match only a non-null object of the correct type.

You can test this code using the following code in `Program.cs`:

C#

```
using System;
using CommercialRegistration;
using ConsumerVehicleRegistration;
using LiveryRegistration;

using toll_calculator;

var tollCalc = new TollCalculator();

var car = new Car();
var taxi = new Taxi();
var bus = new Bus();
var truck = new DeliveryTruck();

Console.WriteLine($"The toll for a car is {tollCalc.CalculateToll(car)}");
Console.WriteLine($"The toll for a taxi is {tollCalc.CalculateToll(taxi)}");
Console.WriteLine($"The toll for a bus is {tollCalc.CalculateToll(bus)}");
Console.WriteLine($"The toll for a truck is
{tollCalc.CalculateToll(truck)}");

try
{
    tollCalc.CalculateToll("this will fail");
}
catch (ArgumentException e)
{
    Console.WriteLine("Caught an argument exception when using the wrong
type");
}
try
{
    tollCalc.CalculateToll(null!);
}
catch (ArgumentNullException e)
{
    Console.WriteLine("Caught an argument exception when using null");
}
```

That code is included in the starter project, but is commented out. Remove the comments, and you can test what you've written.

You're starting to see how patterns can help you create algorithms where the code and the data are separate. The `switch` expression tests the type and produces different values based on the results. That's only the beginning.

Add occupancy pricing

The toll authority wants to encourage vehicles to travel at maximum capacity. They've decided to charge more when vehicles have fewer passengers, and encourage full vehicles by offering lower pricing:

- Cars and taxis with no passengers pay an extra $0.50.
- Cars and taxis with two passengers get a $0.50 discount.
- Cars and taxis with three or more passengers get a $1.00 discount.
- Buses that are less than 50% full pay an extra $2.00.
- Buses that are more than 90% full get a $1.00 discount.

These rules can be implemented using a property pattern in the same switch expression. A property pattern compares a property value to a constant value. The property pattern examines properties of the object once the type has been determined. The single case for a `Car` expands to four different cases:

C#

```
vehicle switch
{
    Car {Passengers: 0} => 2.00m + 0.50m,
    Car {Passengers: 1} => 2.0m,
    Car {Passengers: 2} => 2.0m - 0.50m,
    Car                 => 2.00m - 1.0m,

    // ...
};
```

The first three cases test the type as a `Car`, then check the value of the `Passengers` property. If both match, that expression is evaluated and returned.

You would also expand the cases for taxis in a similar manner:

C#

```
vehicle switch
{
    // ...

    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    // ...
};
```

Next, implement the occupancy rules by expanding the cases for buses, as shown in the following example:

```
C#
```

```
vehicle switch
{
    // ...

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    // ...
};
```

The toll authority isn't concerned with the number of passengers in the delivery trucks. Instead, they adjust the toll amount based on the weight class of the trucks as follows:

- Trucks over 5000 lbs are charged an extra $5.00.
- Light trucks under 3000 lbs are given a $2.00 discount.

That rule is implemented with the following code:

```
C#
```

```
vehicle switch
{
    // ...

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,
};
```

The preceding code shows the `when` clause of a switch arm. You use the `when` clause to test conditions other than equality on a property. When you've finished, you'll have a method that looks much like the following code:

```
C#
```

```
vehicle switch
{
    Car {Passengers: 0}        => 2.00m + 0.50m,
    Car {Passengers: 1}        => 2.0m,
    Car {Passengers: 2}        => 2.0m - 0.50m,
    Car                        => 2.00m - 1.0m,
```

```
    Taxi {Fares: 0}  => 3.50m + 1.00m,
    Taxi {Fares: 1}  => 3.50m,
    Taxi {Fares: 2}  => 3.50m - 0.50m,
    Taxi             => 3.50m - 1.00m,

    Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
2.00m,
    Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
    Bus => 5.00m,

    DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
    DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
    DeliveryTruck => 10.00m,

    { }     => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
    null    => throw new ArgumentNullException(nameof(vehicle))
};
```

Many of these switch arms are examples of **recursive patterns**. For example, `Car { Passengers: 1}` shows a constant pattern inside a property pattern.

You can make this code less repetitive by using nested switches. The `Car` and `Taxi` both have four different arms in the preceding examples. In both cases, you can create a declaration pattern that feeds into a constant pattern. This technique is shown in the following code:

C#

```
public decimal CalculateToll(object vehicle) =>
    vehicle switch
    {
        Car c => c.Passengers switch
        {
            0 => 2.00m + 0.5m,
            1 => 2.0m,
            2 => 2.0m - 0.5m,
            _ => 2.00m - 1.0m
        },

        Taxi t => t.Fares switch
        {
            0 => 3.50m + 1.00m,
            1 => 3.50m,
            2 => 3.50m - 0.50m,
            _ => 3.50m - 1.00m
        },

        Bus b when ((double)b.Riders / (double)b.Capacity) < 0.50 => 5.00m +
```

```
2.00m,
        Bus b when ((double)b.Riders / (double)b.Capacity) > 0.90 => 5.00m -
1.00m,
        Bus b => 5.00m,

        DeliveryTruck t when (t.GrossWeightClass > 5000) => 10.00m + 5.00m,
        DeliveryTruck t when (t.GrossWeightClass < 3000) => 10.00m - 2.00m,
        DeliveryTruck t => 10.00m,

        { }  => throw new ArgumentException(message: "Not a known vehicle
type", paramName: nameof(vehicle)),
        null => throw new ArgumentNullException(nameof(vehicle))
    };
```

In the preceding sample, using a recursive expression means you don't repeat the `Car` and `Taxi` arms containing child arms that test the property value. This technique isn't used for the `Bus` and `DeliveryTruck` arms because those arms are testing ranges for the property, not discrete values.

Add peak pricing

For the final feature, the toll authority wants to add time sensitive peak pricing. During the morning and evening rush hours, the tolls are doubled. That rule only affects traffic in one direction: inbound to the city in the morning, and outbound in the evening rush hour. During other times during the workday, tolls increase by 50%. Late night and early morning, tolls are reduced by 25%. During the weekend, it's the normal rate, regardless of the time. You could use a series of `if` and `else` statements to express this using the following code:

C#

```
public decimal PeakTimePremiumIfElse(DateTime timeOfToll, bool inbound)
{
    if ((timeOfToll.DayOfWeek == DayOfWeek.Saturday) ||
        (timeOfToll.DayOfWeek == DayOfWeek.Sunday))
    {
        return 1.0m;
    }
    else
    {
        int hour = timeOfToll.Hour;
        if (hour < 6)
        {
            return 0.75m;
        }
        else if (hour < 10)
        {
            if (inbound)
```

```
            {
                return 2.0m;
            }
            else
            {
                return 1.0m;
            }
        }
        else if (hour < 16)
        {
            return 1.5m;
        }
        else if (hour < 20)
        {
            if (inbound)
            {
                return 1.0m;
            }
            else
            {
                return 2.0m;
            }
        }
        else // Overnight
        {
            return 0.75m;
        }
    }
}
```

The preceding code does work correctly, but isn't readable. You have to chain through all the input cases and the nested `if` statements to reason about the code. Instead, you'll use pattern matching for this feature, but you'll integrate it with other techniques. You could build a single pattern match expression that would account for all the combinations of direction, day of the week, and time. The result would be a complicated expression. It would be hard to read and difficult to understand. That makes it hard to ensure correctness. Instead, combine those methods to build a tuple of values that concisely describes all those states. Then use pattern matching to calculate a multiplier for the toll. The tuple contains three discrete conditions:

- The day is either a weekday or a weekend.
- The band of time when the toll is collected.
- The direction is into the city or out of the city

The following table shows the combinations of input values and the peak pricing multiplier:

Expand table

Day	Time	Direction	Premium
Weekday	morning rush	inbound	x 2.00
Weekday	morning rush	outbound	x 1.00
Weekday	daytime	inbound	x 1.50
Weekday	daytime	outbound	x 1.50
Weekday	evening rush	inbound	x 1.00
Weekday	evening rush	outbound	x 2.00
Weekday	overnight	inbound	x 0.75
Weekday	overnight	outbound	x 0.75
Weekend	morning rush	inbound	x 1.00
Weekend	morning rush	outbound	x 1.00
Weekend	daytime	inbound	x 1.00
Weekend	daytime	outbound	x 1.00
Weekend	evening rush	inbound	x 1.00
Weekend	evening rush	outbound	x 1.00
Weekend	overnight	inbound	x 1.00
Weekend	overnight	outbound	x 1.00

There are 16 different combinations of the three variables. By combining some of the conditions, you'll simplify the final switch expression.

The system that collects the tolls uses a DateTime structure for the time when the toll was collected. Build member methods that create the variables from the preceding table. The following function uses a pattern matching switch expression to express whether a DateTime represents a weekend or a weekday:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Monday    => true,
        DayOfWeek.Tuesday   => true,
        DayOfWeek.Wednesday => true,
        DayOfWeek.Thursday  => true,
        DayOfWeek.Friday    => true,
```

```
        DayOfWeek.Saturday  => false,
        DayOfWeek.Sunday    => false
    };
```

That method is correct, but it's repetitious. You can simplify it, as shown in the following code:

C#

```
private static bool IsWeekDay(DateTime timeOfToll) =>
    timeOfToll.DayOfWeek switch
    {
        DayOfWeek.Saturday => false,
        DayOfWeek.Sunday => false,
        _ => true
    };
```

Next, add a similar function to categorize the time into the blocks:

C#

```
private enum TimeBand
{
    MorningRush,
    Daytime,
    EveningRush,
    Overnight
}

private static TimeBand GetTimeBand(DateTime timeOfToll) =>
    timeOfToll.Hour switch
    {
        < 6 or > 19 => TimeBand.Overnight,
        < 10 => TimeBand.MorningRush,
        < 16 => TimeBand.Daytime,
        _ => TimeBand.EveningRush,
    };
```

You add a private `enum` to convert each range of time to a discrete value. Then, the `GetTimeBand` method uses relational patterns, and conjunctive or patterns. A relational pattern lets you test a numeric value using `<`, `>`, `<=`, or `>=`. The `or` pattern tests if an expression matches one or more patterns. You can also use an `and` pattern to ensure that an expression matches two distinct patterns, and a `not` pattern to test that an expression doesn't match a pattern.

After you create those methods, you can use another `switch` expression with the **tuple pattern** to calculate the pricing premium. You could build a `switch` expression with all

16 arms:

```C#
public decimal PeakTimePremiumFull(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.MorningRush, true) => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime, true) => 1.50m,
        (true, TimeBand.Daytime, false) => 1.50m,
        (true, TimeBand.EveningRush, true) => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight, true) => 0.75m,
        (true, TimeBand.Overnight, false) => 0.75m,
        (false, TimeBand.MorningRush, true) => 1.00m,
        (false, TimeBand.MorningRush, false) => 1.00m,
        (false, TimeBand.Daytime, true) => 1.00m,
        (false, TimeBand.Daytime, false) => 1.00m,
        (false, TimeBand.EveningRush, true) => 1.00m,
        (false, TimeBand.EveningRush, false) => 1.00m,
        (false, TimeBand.Overnight, true) => 1.00m,
        (false, TimeBand.Overnight, false) => 1.00m,
    };
```

The above code works, but it can be simplified. All eight combinations for the weekend have the same toll. You can replace all eight with the following line:

```C#
(false, _, _) => 1.0m,
```

Both inbound and outbound traffic have the same multiplier during the weekday daytime and overnight hours. Those four switch arms can be replaced with the following two lines:

```C#
(true, TimeBand.Overnight, _) => 0.75m,
(true, TimeBand.Daytime, _)   => 1.5m,
```

The code should look like the following code after those two changes:

```C#
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
```

```
        (true, TimeBand.MorningRush, true)  => 2.00m,
        (true, TimeBand.MorningRush, false) => 1.00m,
        (true, TimeBand.Daytime,     _)     => 1.50m,
        (true, TimeBand.EveningRush, true)  => 1.00m,
        (true, TimeBand.EveningRush, false) => 2.00m,
        (true, TimeBand.Overnight,   _)     => 0.75m,
        (false, _,                   _)     => 1.00m,
    };
```

Finally, you can remove the two rush hour times that pay the regular price. Once you remove those arms, you can replace the `false` with a discard (`_`) in the final switch arm. You'll have the following finished method:

C#

```
public decimal PeakTimePremium(DateTime timeOfToll, bool inbound) =>
    (IsWeekDay(timeOfToll), GetTimeBand(timeOfToll), inbound) switch
    {
        (true, TimeBand.Overnight, _) => 0.75m,
        (true, TimeBand.Daytime, _) => 1.5m,
        (true, TimeBand.MorningRush, true) => 2.0m,
        (true, TimeBand.EveningRush, false) => 2.0m,
        _ => 1.0m,
    };
```

This example highlights one of the advantages of pattern matching: the pattern branches are evaluated in order. If you rearrange them so that an earlier branch handles one of your later cases, the compiler warns you about the unreachable code. Those language rules made it easier to do the preceding simplifications with confidence that the code didn't change.

Pattern matching makes some types of code more readable and offers an alternative to object-oriented techniques when you can't add code to your classes. The cloud is causing data and functionality to live apart. The *shape* of the data and the *operations* on it aren't necessarily described together. In this tutorial, you consumed existing data in entirely different ways from its original function. Pattern matching gave you the ability to write functionality that overrode those types, even though you couldn't extend them.

Next steps

You can download the finished code from the dotnet/samples GitHub repository. Explore patterns on your own and add this technique into your regular coding activities. Learning these techniques gives you another way to approach problems and create new functionality.

See also

- Patterns
- switch expression

Use record types

> **Tip**
>
> **New to developing software?** Start with the Get started tutorials first. You get comfortable with classes, methods, and loops there.
>
> **Experienced in another language?** This tutorial focuses on C# record features you use every day: value equality, positional syntax, and `with` expressions.

In this tutorial, you build a console app that models daily temperatures by using records and record structs.

In this tutorial, you learn how to:

- Declare positional records and record structs.
- Build a small record hierarchy.
- Use compiler-generated equality and formatting.
- Use `with` expressions for nondestructive mutation.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Installation instructions

On Windows, this WinGet configuration file to install all prerequisites. If you already have something installed, WinGet will skip that step.

1. Download the file and double-click to run it.
2. Read the license agreement, type `y`, and select `Enter` when prompted to accept.
3. If you get a flashing User Account Control (UAC) prompt in your Taskbar, allow the installation to continue.

On other platforms, you need to install each of these components separately.

1. Download the recommended installer from the .NET SDK download page and double-click to run it. The download page detects your platform and recommends the latest installer for your platform.
2. Download the latest installer from the Visual Studio Code home page and double click to run it. That page also detects your platform and the link should be correct for your system.
3. Click the "Install" button on the C# DevKit extension page. That opens Visual Studio code, and asks if you want to install or enable the extension. Select "install".

Create the app and your first record

Create a folder for your app, run `dotnet new console`, and open the generated project.

Add a file named `DailyTemperature.cs`, and add a positional `readonly record struct` for temperature values:

C#

```
public readonly record struct DailyTemperature(double HighTemp, double LowTemp)
{
    public double Mean => (HighTemp + LowTemp) / 2.0;
}
```

Add a file named `Program.cs`, and create sample temperature data:

C#

```
private static DailyTemperature[] data = [
    new DailyTemperature(HighTemp: 57, LowTemp: 30),
    new DailyTemperature(60, 35),
    new DailyTemperature(63, 33),
    new DailyTemperature(68, 29),
    new DailyTemperature(72, 47),
    new DailyTemperature(75, 55),
    new DailyTemperature(77, 55),
    new DailyTemperature(72, 58),
    new DailyTemperature(70, 47),
    new DailyTemperature(77, 59),
    new DailyTemperature(85, 65),
    new DailyTemperature(87, 65),
    new DailyTemperature(85, 72),
    new DailyTemperature(83, 68),
    new DailyTemperature(77, 65),
    new DailyTemperature(72, 58),
    new DailyTemperature(77, 55),
    new DailyTemperature(76, 53),
    new DailyTemperature(80, 60),
```

```
    new DailyTemperature(85, 66)
];
```

This syntax gives you concise data modeling with immutable value semantics.

Add behavior to the record struct

In `DailyTemperature.cs`, the record struct already has a computed `Mean` property:

C#

```
public double Mean => (HighTemp + LowTemp) / 2.0;
```

A record struct works well here because each value is small and self-contained.

Build record types for degree-day calculations

> ⓘ Note
>
> **Heating degree-days** and **cooling degree-days** measure how much the daily average temperature deviates from a base temperature (typically 65°F/18°C). Heating degree-days accumulate on cold days when the average is below the base, while cooling degree-days accumulate on warm days when the average is above the base. These calculations help estimate energy consumption for heating or cooling buildings, making them useful for utility companies, building managers, and climate analysis.

Create a file named `DegreeDays.cs` with a hierarchy for heating and cooling degree-day calculations:

C#

```
public abstract record DegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords);

public sealed record HeatingDegreeDays(double BaseTemperature,
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean < BaseTemperature).Sum(s
=> BaseTemperature - s.Mean);
}

public sealed record CoolingDegreeDays(double BaseTemperature,
```

```
IEnumerable<DailyTemperature> TempRecords)
    : DegreeDays(BaseTemperature, TempRecords)
{
    public double DegreeDays => TempRecords.Where(s => s.Mean > BaseTemperature).Sum(s
=> s.Mean - BaseTemperature);
}
```

Now calculate totals from your `Main` method in `Program.cs`:

C#

```
var heatingDegreeDays = new HeatingDegreeDays(65, data);
Console.WriteLine(heatingDegreeDays);

var coolingDegreeDays = new CoolingDegreeDays(65, data);
Console.WriteLine(coolingDegreeDays);
```

The generated `ToString` output is useful for quick diagnostics while you iterate.

Override `PrintMembers` to customize output

When the default output includes too much noise, override `PrintMembers` in the base record:

C#

```
protected virtual bool PrintMembers(StringBuilder stringBuilder)
{
    stringBuilder.Append($"BaseTemperature = {BaseTemperature}");
    return true;
}
```

The override keeps the output focused on the information you need.

Use with expressions for nondestructive mutation

Use `with` to create modified copies without mutating the original record:

C#

```
// Growing degree days measure warming to determine plant growing rates
var growingDegreeDays = coolingDegreeDays with { BaseTemperature = 41 };
Console.WriteLine(growingDegreeDays);
```

Extend that idea to compute rolling totals from slices of your input data:

C#

```
// showing moving accumulation of 5 days using range syntax
List<CoolingDegreeDays> movingAccumulation = new();
int rangeSize = (data.Length > 5) ? 5 : data.Length;
for (int start = 0; start < data.Length - rangeSize; start++)
{
    var fiveDayTotal = growingDegreeDays with { TempRecords = data[start..(start +
rangeSize)] };
    movingAccumulation.Add(fiveDayTotal);
}
Console.WriteLine();
Console.WriteLine("Total degree days in the last five days");
foreach(var item in movingAccumulation)
{
    Console.WriteLine(item);
}
```

This approach is useful when you need transformations while you preserve original values.

Next steps

- Review C# record types for deeper guidance.
- Continue with Object-oriented C# for broader design patterns.
- Explore Converting types to combine records with safe conversion patterns.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/30/2026

How to handle an exception using try/catch

Article • 04/22/2023

The purpose of a try-catch block is to catch and handle an exception generated by working code. Some exceptions can be handled in a `catch` block and the problem solved without the exception being rethrown; however, more often the only thing that you can do is make sure that the appropriate exception is thrown.

Example

In this example, IndexOutOfRangeException isn't the most appropriate exception: ArgumentOutOfRangeException makes more sense for the method because the error is caused by the `index` argument passed in by the caller.

C#

```
static int GetInt(int[] array, int index)
{
    try
    {
        return array[index];
    }
    catch (IndexOutOfRangeException e)  // CS0168
    {
        Console.WriteLine(e.Message);
        // Set IndexOutOfRangeException to the new exception's
InnerException.
        throw new ArgumentOutOfRangeException("index parameter is out of
range.", e);
    }
}
```

Comments

The code that causes an exception is enclosed in the `try` block. A `catch` statement is added immediately after it to handle `IndexOutOfRangeException`, if it occurs. The `catch` block handles the `IndexOutOfRangeException` and throws the more appropriate `ArgumentOutOfRangeException` instead. In order to provide the caller with as much information as possible, consider specifying the original exception as the InnerException

of the new exception. Because the InnerException property is read-only, you must assign it in the constructor of the new exception.

How to execute cleanup code using finally

Article • 04/22/2023

The purpose of a `finally` statement is to ensure that the necessary cleanup of objects, usually objects that are holding external resources, occurs immediately, even if an exception is thrown. One example of such cleanup is calling Close on a FileStream immediately after use instead of waiting for the object to be garbage collected by the common language runtime, as follows:

C#

```
static void CodeWithoutCleanup()
{
    FileStream? file = null;
    FileInfo fileInfo = new FileInfo("./file.txt");

    file = fileInfo.OpenWrite();
    file.WriteByte(0xF);

    file.Close();
}
```

Example

To turn the previous code into a `try-catch-finally` statement, the cleanup code is separated from the working code, as follows.

C#

```
static void CodeWithCleanup()
{
    FileStream? file = null;
    FileInfo? fileInfo = null;

    try
    {
        fileInfo = new FileInfo("./file.txt");

        file = fileInfo.OpenWrite();
        file.WriteByte(0xF);
    }
    catch (UnauthorizedAccessException e)
    {
        Console.WriteLine(e.Message);
```

```
    }
    finally
    {
        file?.Close();
    }
}
```

Because an exception can occur at any time within the `try` block before the `OpenWrite()` call, or the `OpenWrite()` call itself could fail, we aren't guaranteed that the file is open when we try to close it. The `finally` block adds a check to make sure that the FileStream object isn't `null` before you call the Close method. Without the `null` check, the `finally` block could throw its own NullReferenceException, but throwing exceptions in `finally` blocks should be avoided if it's possible.

A database connection is another good candidate for being closed in a `finally` block. Because the number of connections allowed to a database server is sometimes limited, you should close database connections as quickly as possible. If an exception is thrown before you can close your connection, using the `finally` block is better than waiting for garbage collection.

See also

- using statement
- Exception-handling statements

What's new in C# 15

C# 15 includes the following new features. Try these features by using the latest Visual Studio 2026 insiders version or the .NET 11 preview SDK:

- Collection expression arguments
- Union types

C# 15 is the latest C# preview release. .NET 11 preview versions support C# 15. For more information, see C# language versioning.

You can download the latest .NET 11 preview SDK from the .NET downloads page. You can also download Visual Studio 2026 insiders, which includes the .NET 11 preview SDK.

The "What's new in C#" page adds new features when they're available in public preview releases. The working set section of the roslyn feature status page tracks when upcoming features are merged into the main branch.

You can find any breaking changes introduced in C# 15 in our article on breaking changes.

> **Note**
>
> We're interested in your feedback on these features. If you find issues with any of these new features, create a new issue in the dotnet/roslyn repository.

Collection expression arguments

You can pass arguments to the underlying collection's constructor or factory method by using a `with(...)` element as the first element in a collection expression. This feature enables you to specify capacity, comparers, or other constructor parameters directly within the collection expression syntax.

The following example shows how to pass a capacity argument to a `List<T>` constructor and a comparer to a `HashSet<T>`:

C#

```
string[] values = ["one", "two", "three"];

// Pass capacity argument to List<T> constructor
List<string> names = [with(capacity: values.Length * 2), .. values];

// Pass comparer argument to HashSet<T> constructor
HashSet<string> set = [with(StringComparer.OrdinalIgnoreCase), "Hello", "HELLO",
```

```
"hello"];
// set contains only one element because all strings are equal with
OrdinalIgnoreCase
```

To learn more about collection expression arguments, see the language reference article on collection expressions or the feature specification. For information on using collection expression arguments in collection initializers, see Object and Collection Initializers.

Union types

C# 15 introduces *union types*, which represent a value that can be one of several *case types*. Declare a union with the `union` keyword:

C#

```
public record class Cat(string Name);
public record class Dog(string Name);
public record class Bird(string Name);

public union Pet(Cat, Dog, Bird);
```

Unions provide implicit conversions from each case type, and the compiler ensures `switch` expressions are exhaustive across all case types:

C#

```
Pet pet = new Dog("Rex");

string name = pet switch
{
    Dog d => d.Name,
    Cat c => c.Name,
    Bird b => b.Name,
};
```

Union types first appeared in .NET 11 Preview 2. In early .NET 11 previews, the `UnionAttribute` and `IUnion` interface aren't included in the runtime, so you must declare them in your project. Later .NET 11 preview versions include these runtime types. Also, some features from the proposal specification aren't yet implemented, including *union member providers*. Those features are coming in future previews.

For more information, see Union types in the language reference or the feature specification.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Last updated on 03/24/2026

What's new in C# 14

C# 14 includes the following new features. You can try these features using the latest Visual Studio 2026 version or the .NET 10 SDK:

- Extension members
- Null-conditional assignment
- nameof supports unbound generic types
- More implicit conversions for Span<T> and ReadOnlySpan<T>
- Modifiers on simple lambda parameters
- field backed properties
- partial events and constructors
- user-defined compound assignment operators
- new preprocessor directives for File-based apps

C# 14 is the latest C# release. C# 14 is supported on **.NET 10**. For more information, see C# language versioning.

You can download the latest .NET 10 SDK from the .NET downloads page. You can also download Visual Studio 2026, which includes the .NET 10 SDK.

You can find any breaking changes introduced in C# 14 in our article on breaking changes.

> **Note**
>
> We're interested in your feedback on these features. If you find issues with any of these new features, create a new issue in the dotnet/roslyn repository.

Extension members

C# 14 adds new syntax to define *extension members*. The new syntax enables you to declare *extension properties* in addition to extension methods. You can also declare extension members that extend the type, rather than an instance of the type. In other words, these new extension members can appear as static members of the type you extend. These extensions can include user defined operators implemented as static extension methods. The following code example shows an example of the different kinds of extension members you can declare:

C#

```
public static class Enumerable
{
    // Extension block
    extension<TSource>(IEnumerable<TSource> source) // extension members for
```

```
IEnumerable<TSource>
    {
        // Extension property:
        public bool IsEmpty => !source.Any();

        // Extension method:
        public IEnumerable<TSource> Where(Func<TSource, bool> predicate) { ... }
    }

    // extension block, with a receiver type only
    extension<TSource>(IEnumerable<TSource>) // static extension members for
IEnumerable<Source>
    {
        // static extension method:
        public static IEnumerable<TSource> Combine(IEnumerable<TSource> first,
IEnumerable<TSource> second) { ... }

        // static extension property:
        public static IEnumerable<TSource> Identity => Enumerable.Empty<TSource>();

        // static user defined operator:
        public static IEnumerable<TSource> operator + (IEnumerable<TSource> left,
IEnumerable<TSource> right) => left.Concat(right);
    }
}
```

The members in the first extension block are called as though they're instance members of `IEnumerable<TSource>`, for example `sequence.IsEmpty`. The members in the second extension block are called as though they're static members of `IEnumerable<TSource>`, for example `IEnumerable<int>.Identity`.

You can learn more details by reading the article on extension members in the programming guide, the language reference article on the extension keyword, and the feature specification for the new extension members feature.

The `field` keyword

The token `field` enables you to write a property accessor body without declaring an explicit backing field. The token `field` is replaced with a compiler synthesized backing field.

For example, previously, if you wanted to ensure that a `string` property couldn't be set to `null`, you had to declare a backing field and implement both accessors:

C#

```
private string _msg;
public string Message
{
```

```
    get => _msg;
    set => _msg = value ?? throw new ArgumentNullException(nameof(value));
}
```

You can now simplify your code to:

C#

```
public string Message
{
    get;
    set => field = value ?? throw new ArgumentNullException(nameof(value));
}
```

You can declare a body for one or both accessors for a field backed property.

There's a potential breaking change or confusion reading code in types that also include a symbol named `field`. You can use `@field` or `this.field` to disambiguate between the `field` keyword and the identifier, or you can rename the current `field` symbol to provide better distinction.

Implicit span conversions

C# 14 introduces first-class support for System.Span<T> and System.ReadOnlySpan<T> in the language. This support involves new implicit conversions allowing more natural programming with these types.

`Span<T>` and `ReadOnlySpan<T>` are used in many key ways in C# and the runtime. Their introduction improves performance without risking safety. C# 14 recognizes the relationship and supports some conversions between `ReadOnlySpan<T>`, `Span<T>`, and `T[]`. The span types can be extension method receivers, compose with other conversions, and help with generic type inference scenarios.

You can find the list of implicit span conversions in the article on built-in types in the language reference section. You can learn more details by reading the feature specification for First class span types.

Unbound generic types and `nameof`

Beginning with C# 14, the argument to `nameof` can be an unbound generic type. For example, `nameof(List<>)` evaluates to `List`. In earlier versions of C#, only closed generic types, such as `List<int>`, could be used to return the `List` name.

Simple lambda parameters with modifiers

You can add parameter modifiers, such as `scoped`, `ref`, `in`, `out`, or `ref readonly` to lambda expression parameters without specifying the parameter type:

C#

```
delegate bool TryParse<T>(string text, out T result);
// ...
TryParse<int> parse1 = (text, out result) => Int32.TryParse(text, out result);
```

Previously, adding any modifiers was allowed only when the parameter declarations included the types for the parameters. The preceding declaration would require types on all parameters:

C#

```
TryParse<int> parse2 = (string text, out int result) => Int32.TryParse(text, out
result);
```

The `params` modifier still requires an explicitly typed parameter list.

You can read more about these changes in the article on lambda expressions in the C# language reference.

More partial members

You can now declare instance constructors and events as partial members.

Partial constructors and partial events must include exactly one *defining declaration* and one *implementing declaration*.

Only the implementing declaration of a partial constructor can include a constructor initializer: `this()` or `base()`. Only one partial type declaration can include the primary constructor syntax.

The implementing declaration of a partial event must include `add` and `remove` accessors. The defining declaration declares a field-like event.

User defined compound assignment

You can learn more in the feature specification for user-defined compound assignment.

Null-conditional assignment

The null-conditional member access operators, `?.` and `?[]`, can now be used on the left hand side of an assignment or compound assignment.

Before C# 14, you needed to null-check a variable before assigning to a property:

C#

```
if (customer is not null)
{
    customer.Order = GetCurrentOrder();
}
```

You can simplify the preceding code using the `?.` operator:

C#

```
customer?.Order = GetCurrentOrder();
```

The right side of the `=` operator is evaluated only when the left side isn't null. If `customer` is null, the code doesn't call `GetCurrentOrder`.

In addition to assignment, you can use null-conditional member access operators with compound assignment operators (`+=`, `-=`, and others). However, increment and decrement, `++` and `--`, aren't allowed.

You can learn more in the language reference article on the conditional member access and the feature specification for null-conditional assignment.

See also

- What's new in .NET 10

Last updated on 11/19/2025

What's new in C# 13

C# 13 includes the following new features. You can try these features using the latest Visual Studio 2022 version or the .NET 9 SDK:

- params collections
- New lock type and semantics.
- New escape sequence - \e.
- Method group natural type improvements
- Implicit indexer access in object initializers
- Enable ref locals and unsafe contexts in iterators and async methods
- Enable ref struct types to implement interfaces.
- Allow ref struct types as arguments for type parameters in generics.
- Partial properties and indexers are now allowed in `partial` types.
- Overload resolution priority allows library authors to designate one overload as better than others.

Beginning with Visual Studio 17.12, C# 13 includes the field contextual keyword as a preview feature.

C# 13 is supported on **.NET 9**. For more information, see C# language versioning.

You can download the latest .NET 9 SDK from the .NET downloads page. You can also download Visual Studio 2022, which includes the .NET 9 SDK.

You can find any breaking changes introduced in C# 13 in our article on breaking changes.

> **Note**
>
> We're interested in your feedback on these features. If you find issues with any of these new features, create a **new issue** in the **dotnet/roslyn** repository.

`params` collections

The `params` modifier isn't limited to array types. You can now use `params` with any recognized collection type, including System.Span<T>, System.ReadOnlySpan<T>, and types that implement System.Collections.Generic.IEnumerable<T> and have an `Add` method. In addition to concrete types, the interfaces System.Collections.Generic.IEnumerable<T>, System.Collections.Generic.IReadOnlyCollection<T>, System.Collections.Generic.IReadOnlyList<T>, System.Collections.Generic.ICollection<T>, and System.Collections.Generic.IList<T> can also be used.

When an interface type is used, the compiler synthesizes the storage for the arguments supplied. You can learn more in the feature specification for params collections.

For example, method declarations can declare spans as `params` parameters:

C#

```
public void Concat<T>(params ReadOnlySpan<T> items)
{
    for (int i = 0; i < items.Length; i++)
    {
        Console.Write(items[i]);
        Console.Write(" ");
    }
    Console.WriteLine();
}
```

New lock object

The .NET 9 runtime includes a new type for thread synchronization, the System.Threading.Lock type. This type provides better thread synchronization through its API. The Lock.EnterScope() method enters an exclusive scope. The `ref struct` returned from that supports the `Dispose()` pattern to exit the exclusive scope.

The C# lock statement recognizes if the target of the lock is a `Lock` object. If so, it uses the updated API, rather than the traditional API using System.Threading.Monitor. The compiler also recognizes if you convert a `Lock` object to another type and the `Monitor` based code would be generated. You can read more in the feature specification for the new lock object.

This feature allows you to get the benefits of the new library type by changing the type of object you `lock`. No other code needs to change.

New escape sequence

You can use `\e` as a character literal escape sequence for the `ESCAPE` character, Unicode `U+001B`. Previously, you used `\u001b` or `\x1b`. Using `\x1b` wasn't recommended because if the next characters following `1b` were valid hexadecimal digits, those characters became part of the escape sequence.

Method group natural type

This feature makes small optimizations to overload resolution involving method groups. A *method group* is a method and all overloads with the same name. The previous behavior was for the compiler to construct the full set of candidate methods for a method group. If a natural type was needed, the natural type was determined from the full set of candidate methods.

The new behavior is to prune the set of candidate methods at each scope, removing those candidate methods that aren't applicable. Typically, the removed methods are generic methods with the wrong arity, or constraints that aren't satisfied. The process continues to the next outer scope only if no candidate methods are found. This process more closely follows the general algorithm for overload resolution. If all candidate methods found at a given scope don't match, the method group doesn't have a natural type.

You can read the details of the changes in the proposal specification.

Implicit index access

The implicit "from the end" index operator, `^`, is now allowed in an object initializer expression for single-dimension collections. For example, you can now initialize a single-dimension array using an object initializer as shown in the following code:

C#

```
public class TimerRemaining
{
    public int[] buffer { get; set; } = new int[10];
}

var countdown = new TimerRemaining()
{
    buffer =
    {
        [^1] = 0,
        [^2] = 1,
        [^3] = 2,
        [^4] = 3,
        [^5] = 4,
        [^6] = 5,
        [^7] = 6,
        [^8] = 7,
        [^9] = 8,
        [^10] = 9
    }
};
```

The `TimerRemaining` class includes a `buffer` array initialized to a length of 10. The preceding example assigns values to this array using the "from the end" index operator (`^`), effectively creating an array that counts down from 9 to 0.

In versions before C# 13, the `^` operator can't be used in an object initializer. You need to index the elements from the front.

`ref` and `unsafe` in iterators and `async` methods

This feature and the following two features enable `ref struct` types to use new constructs. You won't use these features unless you write your own `ref struct` types. More likely, you see an indirect benefit as System.Span<T> and System.ReadOnlySpan<T> gain more functionality.

Before C# 13, iterator methods (methods that use `yield return`) and `async` methods couldn't declare local `ref` variables, nor could they have an `unsafe` context.

In C# 13, `async` methods can declare `ref` local variables, or local variables of a `ref struct` type. However, those variables can't be accessed across an `await` boundary. Neither can they be accessed across a `yield return` boundary.

This relaxed restriction enables the compiler to allow verifiably safe use of `ref` local variables and `ref struct` types in more places. You can safely use types like System.ReadOnlySpan<T> in these methods. The compiler tells you if you violate safety rules.

In the same fashion, C# 13 allows `unsafe` contexts in iterator methods. However, all `yield return` and `yield break` statements must be in safe contexts.

`allows ref struct`

Before C# 13, `ref struct` types couldn't be declared as the type argument for a generic type or method. Now, generic type declarations can add an anti-constraint, `allows ref struct`. This anti-constraint declares that the type argument supplied for that type parameter can be a `ref struct` type. The compiler enforces ref safety rules on all instances of that type parameter.

For example, you might declare a generic type like the following code:

C#

```
public class C<T> where T : allows ref struct
{
    // Use T as a ref struct:
    public void M(scoped T p)
    {
        // The parameter p must follow ref safety rules
    }
}
```

This enables types such as System.Span<T> and System.ReadOnlySpan<T> to be used with generic algorithms, where applicable. You can learn more in the updates for where and the programming guide article on generic constraints.

`ref struct` interfaces

Before C# 13, `ref struct` types weren't allowed to implement interfaces. Beginning with C# 13, they can. You can declare that a `ref struct` type implements an interface. However, to ensure ref safety rules, a `ref struct` type can't be converted to an interface type. That conversion is a boxing conversion, and could violate ref safety. Explicit interface method declarations in a `ref struct` can be accessed only through a type parameter where that type parameter allows ref struct. Also, `ref struct` types must implement all methods declared in an interface, including those methods with a default implementation.

Learn more in the updates on ref struct types and the addition of the allows ref struct generic constraint.

More partial members

You can declare `partial` properties and `partial` indexers in C# 13. Partial properties and indexers generally follow the same rules as `partial` methods: you create one *declaring declaration* and one *implementing declaration*. The signatures of the two declarations must match. One restriction is that you can't use an auto-property declaration for *implementing* a partial property. Properties that don't declare a body are considered as the *declaring declaration*.

C#

```
public partial class C
{
    // Declaring declaration
    public partial string Name { get; set; }
}

public partial class C
{
    // implementation declaration:
    private string _name;
    public partial string Name
    {
        get => _name;
        set => _name = value;
    }
}
```

You can learn more in the article on partial members.

Overload resolution priority

In C# 13, the compiler recognizes the OverloadResolutionPriorityAttribute to prefer one overload over another. Library authors can use this attribute to ensure that a new, better overload is preferred over an existing overload. For example, you might add a new overload that's more performant. You don't want to break existing code that uses your library, but you want users to update to the new version when they recompile. You can use Overload resolution priority to inform the compiler which overload should be preferred. Overloads with the highest priority are preferred.

This feature is intended for library authors to avoid ambiguity when adding new overloads. Library authors should use care with this attribute to avoid confusion.

The `field` keyword

The field contextual keyword is in C# 13 as a preview feature. The token `field` accesses the compiler synthesized backing field in a property accessor. It enables you to write an accessor body without declaring an explicit backing field in your type declaration. You can declare a body for one or both accessors for a field backed property.

The `field` feature is released as a preview feature. We want to learn from your experiences using it. There's a potential breaking change or confusion reading code in types that also include a field named `field`. You can use `@field` or `this.field` to disambiguate between the `field` keyword and the identifier.

> **Important**
>
> You should be careful using the `field` keyword feature in a class that has a field named `field`. The new `field` keyword shadows a field named `field` in the scope of a property accessor. You can either change the name of the `field` variable, or use the `@` token to reference the `field` identifier as `@field`. You can learn more by reading the feature specification for the field keyword.

If you try this feature and have feedback, add it to the feature issue in the `csharplang` repository.

See also

- What's new in .NET 9

Last updated on 11/18/2025

What's new in C# 12

Article • 06/04/2024

C# 12 includes the following new features. You can try these features using the latest Visual Studio 2022 version or the .NET 8 SDK.

- Primary constructors - Introduced in Visual Studio 2022 version 17.6 Preview 2.
- Collection expressions - Introduced in Visual Studio 2022 version 17.7 Preview 5.
- Inline arrays - Introduced in Visual Studio 2022 version 17.7 Preview 3.
- Optional parameters in lambda expressions - Introduced in Visual Studio 2022 version 17.5 Preview 2.
- ref readonly parameters - Introduced in Visual Studio 2022 version 17.8 Preview 2.
- Alias any type - Introduced in Visual Studio 2022 version 17.6 Preview 3.
- Experimental attribute - Introduced in Visual Studio 2022 version 17.7 Preview 3.
- Interceptors - *Preview feature* Introduced in Visual Studio 2022 version 17.7 Preview 3.

C# 12 is supported on **.NET 8**. For more information, see C# language versioning.

You can download the latest .NET 8 SDK from the .NET downloads page. You can also download Visual Studio 2022, which includes the .NET 8 SDK.

> **Note**
>
> We're interested in your feedback on these features. If you find issues with any of these new features, create a new issue in the dotnet/roslyn repository.

Primary constructors

You can now create primary constructors in any `class` and `struct`. Primary constructors are no longer restricted to `record` types. Primary constructor parameters are in scope for the entire body of the class. To ensure that all primary constructor parameters are definitely assigned, all explicitly declared constructors must call the primary constructor using `this()` syntax. Adding a primary constructor to a `class` prevents the compiler from declaring an implicit parameterless constructor. In a `struct`, the implicit

parameterless constructor initializes all fields, including primary constructor parameters to the 0-bit pattern.

The compiler generates public properties for primary constructor parameters only in `record` types, either `record class` or `record struct` types. Nonrecord classes and structs might not always want this behavior for primary constructor parameters.

You can learn more about primary constructors in the tutorial for exploring primary constructors and in the article on instance constructors.

Collection expressions

Collection expressions introduce a new terse syntax to create common collection values. Inlining other collections into these values is possible using a spread element `..e`.

Several collection-like types can be created without requiring external BCL support. These types are:

- Array types, such as `int[]`.
- System.Span<T> and System.ReadOnlySpan<T>.
- Types that support collection initializers, such as System.Collections.Generic.List<T>.

The following examples show uses of collection expressions:

C#

```
// Create an array:
int[] a = [1, 2, 3, 4, 5, 6, 7, 8];

// Create a list:
List<string> b = ["one", "two", "three"];

// Create a span
Span<char> c  = ['a', 'b', 'c', 'd', 'e', 'f', 'h', 'i'];

// Create a jagged 2D array:
int[][] twoD = [[1, 2, 3], [4, 5, 6], [7, 8, 9]];

// Create a jagged 2D array from variables:
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[][] twoDFromVariables = [row0, row1, row2];
```

The *spread element*, `..e` in a collection expression adds all the elements in that expression. The argument must be a collection type. The following examples show how the spread element works:

C#

```
int[] row0 = [1, 2, 3];
int[] row1 = [4, 5, 6];
int[] row2 = [7, 8, 9];
int[] single = [.. row0, .. row1, .. row2];
foreach (var element in single)
{
    Console.Write($"{element}, ");
}
// output:
// 1, 2, 3, 4, 5, 6, 7, 8, 9,
```

The spread element evaluates each element of the enumerations expression. Each element is included in the output collection.

You can use collection expressions anywhere you need a collection of elements. They can specify the initial value for a collection or be passed as arguments to methods that take collection types. You can learn more about collection expressions in the language reference article on collection expressions or the feature specification.

`ref readonly` parameters

C# added `in` parameters as a way to pass readonly references. `in` parameters allow both variables and values, and can be used without any annotation on arguments.

The addition of `ref readonly` parameters enables more clarity for APIs that might be using `ref` parameters or `in` parameters:

- APIs created before `in` was introduced might use `ref` even though the argument isn't modified. Those APIs can be updated with `ref readonly`. It won't be a breaking change for callers, as would be if the `ref` parameter was changed to `in`. An example is System.Runtime.InteropServices.Marshal.QueryInterface.
- APIs that take an `in` parameter, but logically require a variable. A value expression doesn't work. An example is System.ReadOnlySpan<T>.ReadOnlySpan<T>(T).
- APIs that use `ref` because they require a variable, but don't mutate that variable. An example is System.Runtime.CompilerServices.Unsafe.IsNullRef.

To learn more about `ref readonly` parameters, see the article on parameter modifiers in the language reference, or the ref readonly parameters feature specification.

Default lambda parameters

You can now define default values for parameters on lambda expressions. The syntax and rules are the same as adding default values for arguments to any method or local function.

You can learn more about default parameters on lambda expressions in the article on lambda expressions.

Alias any type

You can use the `using` alias directive to alias any type, not just named types. That means you can create semantic aliases for tuple types, array types, pointer types, or other unsafe types. For more information, see the feature specification. For an example refactoring walkthrough, see Refactor your code using alias any type on the .NET blog.

Inline arrays

Inline arrays are used by the runtime team and other library authors to improve performance in your apps. Inline arrays enable a developer to create an array of fixed size in a `struct` type. A struct with an inline buffer should provide performance characteristics similar to an unsafe fixed size buffer. You likely won't declare your own inline arrays, but you use them transparently when they're exposed as System.Span<T> or System.ReadOnlySpan<T> objects from runtime APIs.

An *inline array* is declared similar to the following `struct`:

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct Buffer
{
    private int _element0;
}
```

You use them like any other array:

C#

```
var buffer = new Buffer();
for (int i = 0; i < 10; i++)
{
    buffer[i] = i;
```

```
}

foreach (var i in buffer)
{
    Console.WriteLine(i);
}
```

The difference is that the compiler can take advantage of known information about an inline array. You likely consume inline arrays as you would any other array. For more information on how to declare inline arrays, see the language reference on struct types.

Experimental attribute

Types, methods, or assemblies can be marked with the System.Diagnostics.CodeAnalysis.ExperimentalAttribute to indicate an experimental feature. The compiler issues a warning if you access a method or type annotated with the ExperimentalAttribute. All types included in an assembly marked with the `Experimental` attribute are experimental. You can read more in the article on General attributes read by the compiler, or the feature specification.

Interceptors

> ⚠ **Warning**
>
> Interceptors are an experimental feature, available in preview mode with C# 12. The feature may be subject to breaking changes or removal in a future release. Therefore, it is not recommended for production or released applications.
>
> In order to use interceptors, the user project must specify the property `<InterceptorsPreviewNamespaces>`. This is a list of namespaces which are allowed to contain interceptors.
>
> For example:
> `<InterceptorsPreviewNamespaces>$(InterceptorsPreviewNamespaces);Microsoft.AspNetCore.Http.Generated;MyLibrary.Generated</InterceptorsPreviewNamespaces>`

An *interceptor* is a method that can declaratively substitute a call to an *interceptable* method with a call to itself at compile time. This substitution occurs by having the interceptor declare the source locations of the calls that it intercepts. Interceptors provide a limited facility to change the semantics of existing code by adding new code to a compilation, for example in a source generator.

You use an *interceptor* as part of a source generator to modify, rather than add code to an existing source compilation. The source generator substitutes calls to an interceptable method with a call to the *interceptor* method.

If you're interested in experimenting with interceptors, you can learn more by reading the feature specification. If you use the feature, make sure to stay current with any changes in the feature specification for this experimental feature. If the feature is finalized, we'll add more guidance on this site.

See also

- What's new in .NET 8

Learn about any breaking changes in the C# compiler

Article • 05/22/2025

To find breaking changes since the C# 10 release, see Breaking changes in Roslyn after .NET 6.0.100 through .NET 7.0.100.

The Roslyn team maintains a list of breaking changes in the C# and Visual Basic compilers. You can find information on those changes at these links on their GitHub repository:

- Breaking changes in Roslyn in C# 10.0/.NET 6
- Breaking changes in Roslyn after .NET 5
- Breaking changes in VS2019 version 16.8 introduced with .NET 5 and C# 9.0
- Breaking changes in VS2019 Update 1 and beyond compared to VS2019
- Breaking changes since VS2017 (C# 7)
- Breaking changes in Roslyn 3.0 (VS2019) from Roslyn 2.* (VS2017)
- Breaking changes in Roslyn 2.0 (VS2017) from Roslyn 1.* (VS2015) and native C# compiler (VS2013 and previous).
- Breaking changes in Roslyn 1.0 (VS2015) from the native C# compiler (VS2013 and previous).
- Unicode version change in C# 6

The history of C#

This article provides a history of each major release of the C# language. The C# team is continuing to innovate and add new features. Detailed language feature status, including features considered for upcoming releases can be found on the dotnet/roslyn repository on GitHub. To find when a particular feature was added to the language, consult the C# version history file in the dotnet/csharplang repository on GitHub.

> **Important**
>
> The C# language relies on types and methods in what the C# specification defines as a *standard library* for some of the features. The .NET platform delivers those types and methods in a number of packages. One example is exception processing. Every `throw` statement or expression is checked to ensure the object being thrown is derived from **Exception**. Similarly, every `catch` is checked to ensure that the type being caught is derived from **Exception**. Each version may add new requirements. To use the latest language features in older environments, you may need to install specific libraries. These dependencies are documented in the page for each specific version. You can learn more about the **relationships between language and library** for background on this dependency.

C# version 14

Released November 2025

C# 14 includes the following new features:

- Extension members
- Null-conditional assignment
- nameof supports unbound generic types
- More implicit conversions for Span<T> and ReadOnlySpan<T>
- Modifiers on simple lambda parameters
- field backed properties
- partial events and constructors
- user-defined compound assignment operators

C# version 13

Released November 2024

C# 13 includes the following new features:

- `params` collections: the `params` modifier isn't limited to array types. You can now use `params` with any recognized collection type, including `Span<T>`, and interface types.
- New `lock` type and semantics: If the target of a `lock` statement is a System.Threading.Lock, compiler generates code to use the Lock.EnterScope() method to enter an exclusive scope. The `ref struct` returned from that supports the `Dispose()` pattern to exit the exclusive scope.
- New escape sequence - `\e`: You can use `\e` as a character literal escape sequence for the `ESCAPE` character, Unicode `U+001B`.
- Small optimizations to overload resolution involving method groups.
- Implicit indexer access in object initializers: The implicit "from the end" index operator, `^`, is now allowed in an object initializer expression.
- You can use `ref` locals and `unsafe` contexts in iterators and async methods.
- You can use `ref struct` types to implement interfaces.
- You can allow `ref struct` types as arguments for type parameters in generics.
- Partial properties and indexers are now allowed in `partial` types.
- Overload resolution priority allows library authors to designate one overload as better than others.

C# version 12

Released November 2023

The following features were added in C# 12:

- Primary constructors - You can create primary constructors in any `class` or `struct` type.
- Collection expressions - A new syntax to specify collection expressions, including the spread element, (`..e`), to expand any collection.
- Inline arrays - Inline arrays enable you to create an array of fixed size in a `struct` type.
- Optional parameters in lambda expressions - You can define default values for parameters on lambda expressions.
- ref readonly parameters - `ref readonly` parameters enables more clarity for APIs that might be using `ref` parameters or `in` parameters.
- Alias any type - You can use the `using` alias directive to alias any type, not just named types.
- Experimental attribute - Indicate an experimental feature.

And, Interceptors - was released as a *Preview feature*.

Overall, C# 12 provides new features that make you more productive writing C# code. Syntax you already knew is available in more places. Other syntax enables consistency for related concepts.

C# version 11

Released November 2022

The following features were added in C# 11:

- Raw string literals.
- Generic math support.
- Generic attributes.
- UTF-8 string literals.
- Newlines in string interpolation expressions.
- List patterns.
- File-local types.
- Required members.
- Auto-default structs.
- Pattern match `Span<char>` on a constant `string`.
- Extended `nameof` scope.
- The `nint` and `uint` keywords alias System.IntPtr and System.UIntPtrSystem.UIntPtr, respectively.
- ref fields and scoped ref
- Improved method group conversion to delegate.
- Warning wave 7

C# 11 introduces *generic math* and several features that support that goal. You can write numeric algorithms once for all number types. There's more features to make working with `struct` types easier, like required members and auto-default structs. Working with strings gets easier with Raw string literals, newline in string interpolations, and UTF-8 string literals. Features like file local types enable source generators to be simpler. Finally, list patterns add more support for pattern matching.

C# version 10

Released November 2021

C# 10 adds the following features and enhancements to the C# language:

- Record structs

- Improvements of structure types
- Interpolated string handlers
- global using directives
- File-scoped namespace declaration
- Extended property patterns
- Lambda expressions can have a natural type, where the compiler can infer a delegate type from the lambda expression or method group.
- Lambda expressions can declare a return type when the compiler can't infer it.
- Attributes can be applied to lambda expressions.
- In C# 10, `const` strings can be initialized using string interpolation if all the placeholders are themselves constant strings.
- In C# 10, you can add the `sealed` modifier when you override `ToString` in a record type.
- Warnings for definite assignment and null-state analysis are more accurate.
- Allow both assignment and declaration in the same deconstruction.
- Allow AsyncMethodBuilder attribute on methods
- CallerArgumentExpression attribute
- C# 10 supports a new format for the `#line` pragma.

C# 10 continues work on themes of removing ceremony, separating data from algorithms, and improved performance for the .NET Runtime.

Many of the features mean you type less code to express the same concepts. *Record structs* synthesize many of the same methods that *record classes* do. Structs and anonymous types support *with expressions*. *Global using directives* and *file scoped namespace declarations* mean you express dependencies and namespace organization more clearly. *Lambda improvements* make it easier to declare lambda expressions where they're used. New property patterns and deconstruction improvements create more concise code.

The new interpolated string handlers and `AsyncMethodBuilder` behavior can improve performance. These language features were applied in the .NET Runtime to achieve performance improvements in .NET 6.

C# 10 also marks more of a shift to the yearly cadence for .NET releases. Because not every feature can be completed in a yearly timeframe, you can try a couple of "preview" features in C# 10. Both *generic attributes* and *static abstract members in interfaces* can be used, but these preview features might change before their final release.

C# version 9

Released November 2020

C# 9 was released with .NET 5. It's the default language version for any assembly that targets the .NET 5 release. It contains the following new and enhanced features:

- Records
- Init only setters
- Top-level statements
- Pattern matching enhancements: relational patterns and logical patterns
- Performance and interop
  - Native sized integers
  - Function pointers
  - Suppress emitting localsinit flag
  - Module initializers
  - New features for partial methods
- Fit and finish features
  - Target-typed new expressions
  - static anonymous functions
  - Target-typed conditional expressions
  - Covariant return types
  - Extension GetEnumerator support for foreach loops
  - Lambda discard parameters
  - Attributes on local functions

C# 9 continues three of the themes from previous releases: removing ceremony, separating data from algorithms, and providing more patterns in more places.

Top level statements means your main program is simpler to read. There's less need for ceremony: a namespace, a `Program` class, and `static void Main()` are all unnecessary.

The introduction of records provides a concise syntax for reference types that follow value semantics for equality. You use these types to define data containers that typically define minimal behavior. Init-only setters provide the capability for nondestructive mutation (`with` expressions) in records. C# 9 also adds covariant return types so that derived records can override virtual methods and return a type derived from the base method's return type.

The pattern matching capabilities expanded in several ways. Numeric types now support *range patterns*. Patterns can be combined using `and`, `or`, and `not` patterns. Parentheses can be added to clarify more complex patterns:

C# 9 includes new pattern matching improvements:

- ***Type patterns*** match an object matches a particular type
- ***Parenthesized patterns*** enforce or emphasize the precedence of pattern combinations
- ***Conjunctive `and` patterns*** require both patterns to match

- ***Disjunctive `or` patterns*** require either pattern to match
- ***Negated `not` patterns*** require that a pattern doesn't match
- ***Relational patterns*** require the input be less than, greater than, less than or equal, or greater than or equal to a given constant

These patterns enrich the syntax for patterns. One of the most common uses is a new syntax for a null check:

```C#
if (e is not null)
{
    // ...
}
```

Any of these patterns can be used in any context where patterns are allowed: `is` pattern expressions, `switch` expressions, nested patterns, and the pattern of a `switch` statement's `case` label.

Another set of features supports high-performance computing in C#:

- The `nint` and `nuint` types model the native-size integer types on the target CPU.
- Function pointers provide delegate-like functionality while avoiding the allocations necessary to create a delegate object.
- The `localsinit` instruction can be omitted to save instructions.

Performance and interop

Another set of improvements supports scenarios where *code generators* add functionality:

- Module initializers are methods that the runtime calls when an assembly loads.
- Partial methods support new accessibly modifiers and non-void return types. In those cases, an implementation must be provided.

Fit and finish features

C# 9 adds many other small features that improve developer productivity, both writing and reading code:

- Target-type `new` expressions
- `static` anonymous functions
- Target-type conditional expressions
- Extension `GetEnumerator()` support for `foreach` loops

- Lambda expressions can declare discard parameters
- Attributes can be applied to local functions

The C# 9 release continues the work to keep C# a modern, general-purpose programming language. Features continue to support modern workloads and application types.

C# version 8.0

Released September 2019

C# 8.0 is the first major C# release that specifically targets .NET Core. Some features rely on new Common Language Runtime (CLR) capabilities, others on library types added only in .NET Core. C# 8.0 adds the following features and enhancements to the C# language:

- Readonly members
- Default interface members
- Pattern matching enhancements:
  - Switch expressions
  - Property patterns
  - Tuple patterns
  - Positional patterns
- Using declarations
- Static local functions
- Disposable ref structs
- Nullable reference types
- Asynchronous streams
- Indices and ranges
- Null-coalescing assignment
- Unmanaged constructed types
- Stackalloc in nested expressions
- Enhancement of interpolated verbatim strings

Default interface members require enhancements in the CLR. Those features were added in the CLR for .NET Core 3.0. Ranges, indexes, and asynchronous streams require new types in the .NET Core 3.0 libraries. Nullable reference types, while implemented in the compiler, is much more useful when libraries are annotated to provide semantic information regarding the null state of arguments and return values. Those annotations are being added in the .NET Core libraries.

C# version 7.3

Released May 2018

There are two main themes to the C# 7.3 release. One theme provides features that enable safe code to be as performant as unsafe code. The second theme provides incremental improvements to existing features. New compiler options were also added in this release.

The following new features support the theme of better performance for safe code:

- You can access fixed fields without pinning.
- You can reassign `ref` local variables.
- You can use initializers on `stackalloc` arrays.
- You can use `fixed` statements with any type that supports a pattern.
- You can use more generic constraints.

The following enhancements were made to existing features:

- You can test `==` and `!=` with tuple types.
- You can use expression variables in more locations.
- You can attach attributes to the backing field of automatically implemented properties.
- Method resolution when arguments differ by `in` was improved.
- Overload resolution now has fewer ambiguous cases.

The new compiler options are:

- -publicsign to enable Open Source Software (OSS) signing of assemblies.
- -pathmap to provide a mapping for source directories.

C# version 7.2

Released November 2017

C# 7.2 added several small language features:

- Initializers on `stackalloc` arrays.
- Use `fixed` statements with any type that supports a pattern.
- Access fixed fields without pinning.
- Reassign `ref` local variables.
- Declare `readonly struct` types, to indicate that a struct is immutable and should be passed as an `in` parameter to its member methods.
- Add the `in` modifier on parameters, to specify that an argument is passed by reference but not modified by the called method.
- Use the `ref readonly` modifier on method returns, to indicate that a method returns its value by reference but doesn't allow writes to that object.

- Declare `ref struct` types, to indicate that a struct type accesses managed memory directly and must always be stack allocated.
- Use more generic constraints.
- Non-trailing named arguments:
  - Positional arguments can follow named arguments.
- Leading underscores in numeric literals:
  - Numeric literals can now have leading underscores before any printed digits.
- private protected access modifier:
  - The `private protected` access modifier enables access for derived classes in the same assembly.
- Conditional `ref` expressions:
  - The result of a conditional expression (`?:`) can now be a reference.

C# version 7.1

Released August 2017

C# started releasing *point releases* with C# 7.1. This version added the language version selection configuration element, three new language features, and new compiler behavior.

The new language features in this release are:

- async Main method
  - The entry point for an application can have the `async` modifier.
- default literal expressions
  - You can use default literal expressions in default value expressions when the target type can be inferred.
- Inferred tuple element names
  - The names of tuple elements can be inferred from tuple initialization in many cases.
- Pattern matching on generic type parameters
  - You can use pattern match expressions on variables whose type is a generic type parameter.

Finally, the compiler has two options -refout and -refonly that control reference assembly generation.

C# version 7.0

Released March 2017

C# version 7.0 was released with Visual Studio 2017. This version has some evolutionary and cool stuff in the vein of C# 6.0. Here are some of the new features:

- Out variables
- Tuples and deconstruction
- Pattern matching
- Local functions
- Expanded expression bodied members
- Ref locals
- Ref returns

Other features included:

- Discards
- Binary Literals and Digit Separators
- Throw expressions

All of these features offer new capabilities for developers and the opportunity to write cleaner code than ever. A highlight is condensing the declaration of variables to use with the `out` keyword and by allowing multiple return values via tuple. .NET Core now targets any operating system and has its eyes firmly on the cloud and on portability. These new capabilities certainly occupy the language designers' thoughts and time, in addition to coming up with new features.

C# version 6.0

Released July 2015

Version 6.0, released with Visual Studio 2015, released many smaller features that made C# programming more productive. Here are some of them:

- Static imports
- Exception filters
- Auto-property initializers
- Expression bodied members
- Null propagator
- String interpolation
- nameof operator

Other new features include:

- Index initializers
- Await in catch/finally blocks

- Default values for getter-only properties

If you look at these features together, you see an interesting pattern. In this version, C# started to eliminate language boilerplate to make code more terse and readable. So for fans of clean, simple code, this language version was a huge win.

They did one other thing along with this version, though it's not a traditional language feature in itself. They released Roslyn the compiler as a service. The C# compiler is now written in C#, and you can use the compiler as part of your programming efforts.

C# version 5.0

Released August 2012

C# version 5.0, released with Visual Studio 2012, was a focused version of the language. Nearly all of the effort for that version went into another groundbreaking language concept: the `async` and `await` model for asynchronous programming. Here's the major features list:

- Asynchronous members
- Caller info attributes

The caller info attribute lets you easily retrieve information about the context in which you're running without resorting to a ton of boilerplate reflection code. It has many uses in diagnostics and logging tasks.

But `async` and `await` are the real stars of this release. When these features came out in 2012, C# changed the game again by baking asynchrony into the language as a first-class participant.

C# version 4.0

Released April 2010

C# version 4.0, released with Visual Studio 2010, introduced some interesting new features:

- Dynamic binding
- Named/optional arguments
- Generic covariant and contravariant
- Embedded interop types

Embedded interop types eased the deployment pain of creating COM interop assemblies for your application. Generic covariance and contravariance give you more power to use generics, but they're a bit academic and probably most appreciated by framework and library authors.

Named and optional parameters let you eliminate many method overloads and provide convenience. But none of those features are exactly paradigm altering.

The major feature was the introduction of the `dynamic` keyword. The `dynamic` keyword introduced into C# version 4.0 the ability to override the compiler on compile-time typing. By using the dynamic keyword, you can create constructs similar to dynamically typed languages like JavaScript. You can create a `dynamic x = "a string"` and then add six to it, leaving it up to the runtime to sort out what should happen next.

Dynamic binding gives you the potential for errors but also great power within the language.

C# version 3.0

Released November 2007

C# version 3.0 came in late 2007, along with Visual Studio 2008, though the full boat of language features would actually come with .NET Framework version 3.5. This version marked a major change in the growth of C#. It established C# as a truly formidable programming language. Let's take a look at some major features in this version:

- Auto-implemented properties
- Anonymous types
- Query expressions
- Lambda expressions
- Expression trees
- Extension methods
- Implicitly typed local variables
- Partial methods
- Object and collection initializers

In retrospect, many of these features seem both inevitable and inseparable. They all fit together strategically. This C# version's killer feature was the query expression, also known as Language-Integrated Query (LINQ).

A more nuanced view examines expression trees, lambda expressions, and anonymous types as the foundation upon which LINQ is constructed. But, in either case, C# 3.0 presented a revolutionary concept. C# 3.0 began to lay the groundwork for turning C# into a hybrid Object-Oriented / Functional language.

Specifically, you could now write SQL-style, declarative queries to perform operations on collections, among other things. Instead of writing a `for` loop to compute the average of a list

of integers, you could now do that as simply as `list.Average()`. The combination of query expressions and extension methods made a list of integers a whole lot smarter.

C# version 2.0

Released November 2005

Let's take a look at some major features of C# 2.0, released in 2005, along with Visual Studio 2005:

- Generics
- Partial types
- Anonymous methods
- Nullable value types
- Iterators
- Covariance and contravariance

Other C# 2.0 features added capabilities to existing features:

- Getter/setter separate accessibility
- Method group conversions (delegates)
- Static classes
- Delegate inference

While C# began as a generic Object-Oriented (OO) language, C# version 2.0 changed that in a hurry. With generics, types and methods can operate on an arbitrary type while still retaining type safety. For instance, having a List<T> lets you have `List<string>` or `List<int>` and perform type-safe operations on those strings or integers while you iterate through them. Using generics is better than creating a `ListInt` type that derives from `ArrayList` or casting from `Object` for every operation.

C# version 2.0 brought iterators. To put it succinctly, iterators let you examine all the items in a `List` (or other Enumerable types) with a `foreach` loop. Having iterators as a first-class part of the language dramatically enhanced readability of the language and people's ability to reason about the code.

C# version 1.2

Released April 2003

C# version 1.2 shipped with Visual Studio .NET 2003. It contained a few small enhancements to the language. Most notable is that starting with this version, the code generated in a `foreach`

loop called Dispose on an IEnumerator when that IEnumerator implemented IDisposable.

C# version 1.0

Released January 2002

When you go back and look, C# version 1.0, released with Visual Studio .NET 2002, looked a lot like Java. As part of its stated design goals for ECMA, it sought to be a "simple, modern, general-purpose object-oriented language." At the time, looking like Java meant it achieved those early design goals.

But if you look back on C# 1.0 now, you'd find yourself a little dizzy. It lacked the built-in async capabilities and some of the slick functionality around generics you take for granted. As a matter of fact, it lacked generics altogether. And LINQ? Not available yet. Those additions would take some years to come out.

C# version 1.0 looked stripped of features, compared to today. You'd find yourself writing some verbose code. But yet, you have to start somewhere. C# version 1.0 was a viable alternative to Java on the Windows platform.

The major features of C# 1.0 included:

- Classes
- Structs
- Interfaces
- Events
- Properties
- Delegates
- Operators and expressions
- Statements
- Attributes

Article originally published on the NDepend blog, courtesy of Erik Dietrich and Patrick Smacchia.

Last updated on 11/18/2025

Relationships between language features and library types

Article • 10/26/2023

The C# language definition requires a standard library to have certain types and certain accessible members on those types. The compiler generates code that uses these required types and members for many different language features. For this reason, C# versions are supported only for the corresponding .NET version and newer. That ensures the correct run-time behavior and the availability of all required types and members.

This dependency on standard library functionality has been part of the C# language since its first version. In that version, examples included:

- Exception - used for all compiler-generated exceptions.
- String - synonym of `string`.
- Int32 - synonym of `int`.

That first version was simple: the compiler and the standard library shipped together, and there was only one version of each.

Subsequent versions of C# have occasionally added new types or members to the dependencies. Examples include: INotifyCompletion, CallerFilePathAttribute, and CallerMemberNameAttribute. C# 7.0 added a dependency on ValueTuple to implement the tuples language feature. C# 8 requires System.Index and System.Range for ranges and indexes, among other features. Each new version might add additional requirements.

The language design team works to minimize the surface area of the types and members required in a compliant standard library. That goal is balanced against a clean design where new library features are incorporated seamlessly into the language. There will be new features in future versions of C# that require new types and members in a standard library. C# compiler tools are now decoupled from the release cycle of the .NET libraries on supported platforms.

Version and update considerations for C# developers

Article • 06/29/2023

Compatibility is an important goal as new features are added to the C# language. In almost all cases, existing code can be recompiled with a new compiler version without any issue. The .NET runtime team also has a goal to ensure compatibility for updated libraries. In almost all cases, when your app is launched from an updated runtime with updated libraries, the behavior is exactly the same as with previous versions.

The language version used to compile your app typically matches the runtime target framework moniker (TFM) referenced in your project. For more information on changing the default language version, see the article titled configure your language version. This default behavior ensures maximum compatibility.

When *breaking changes* are introduced, they're classified as:

- *Binary breaking change*: A binary breaking change causes different behavior, including possibly crashing, in your application or library when launched using a new runtime. You must recompile your app to incorporate these changes. The existing binary won't function correctly.
- *Source breaking change*: A source breaking change changes the meaning of your source code. You need to make source code edits before compiling your application with the latest language version. Your existing binary will run correctly with the newer host and runtime. Note that for language syntax, a *source breaking change* is also a *behavioral change*, as defined in the runtime breaking changes.

When a binary breaking change affects your app, you must recompile your app, but you don't need to edit any source code. When a source breaking change affects your app, the existing binary still runs correctly in environments with the updated runtime and libraries. However, you must make source changes to recompile with the new language version and runtime. If a change is both source breaking and binary breaking, you must recompile your application with the latest version and make source updates.

Because of the goal to avoid breaking changes by the C# language team and runtime team, updating your application is typically a matter of updating the TFM and rebuilding the app. However, for libraries that are distributed publicly, you should carefully evaluate your policy for supported TFMs and supported language versions. You may be creating a new library with features found in the latest version and need to ensure apps built using previous versions of the compiler can use it. Or you may be upgrading an existing library and many of your users might not have upgraded versions yet.

Introducing breaking changes in your libraries

When you adopt new language features in your library's public API, you should evaluate if adopting the feature introduces either a binary or source breaking change for the users of your library. Any changes to your internal implementation that don't appear in the `public` or `protected` interfaces are compatible.

> **Note**
>
> If you use the System.Runtime.CompilerServices.InternalsVisibleToAttribute to enable types to see internal members, the internal members can introduce breaking changes.

A *binary breaking change* requires your users to recompile their code in order to use the new version. For example, consider this public method:

C#

```
public double CalculateSquare(double value) => value * value;
```

If you add the `in` modifier to the method, that's a binary breaking change:

C#

```
public double CalculateSquare(in double value) => value * value;
```

Users must recompile any application that uses the `CalculateSquare` method for the new library to work correctly.

A *source breaking change* requires your users to change their code before they recompile. For example, consider this type:

C#

```
public class Person
{
    public string FirstName { get; }
    public string LastName { get; }

    public Person(string firstName, string lastName) => (FirstName,
LastName) = (firstName, lastName);
```

```
    // other details omitted
}
```

In a newer version, you'd like to take advantage of the synthesized members generated for `record` types. You make the following change:

```
C#
```

```
public record class Person(string FirstName, string LastName);
```

The previous change requires changes for any type derived from `Person`. All those declarations must add the `record` modifier to their declarations.

Impact of breaking changes

When you add a *binary breaking change* to your library, you force all projects that use your library to recompile. However, none of the source code in those projects needs to change. As a result, the impact of the breaking change is reasonably small for each project.

When you make a *source breaking change* to your library, you require all projects to make source changes in order to use your new library. If the necessary change requires new language features, you force those projects to upgrade to the same language version and TFM you're now using. You've required more work for your users, and possibly forced them to upgrade as well.

The impact of any breaking change you make depends on the number of projects that have a dependency on your library. If your library is used internally by a few applications, you can react to any breaking changes in all impacted projects. However, if your library is publicly downloaded, you should evaluate the potential impact and consider alternatives:

- You might add new APIs that parallel existing APIs.
- You might consider parallel builds for different TFMs.
- You might consider multi-targeting.

Tutorial: Explore extension members in C# 14

10/08/2025

C# 14 introduces extension members, an enhancement to the existing extension methods. Extension members enable you to add properties and operators. You can also extend types in addition to instances of types. This capability allows you to create more natural and expressive APIs when extending types you don't control.

In this tutorial, you explore extension members by enhancing the `System.Drawing.Point` type with mathematical operations, coordinate transformations, and utility properties. You learn how to migrate existing extension methods to the new extension member syntax and understand when to use each approach.

In this tutorial, you:

- ✓ Create extension members with static properties and operators.
- ✓ Implement coordinate transformations using extension members.
- ✓ Migrate traditional extension methods to extension member syntax.
- ✓ Compare extension members with traditional extension methods.

Prerequisites

- The .NET 10 preview SDK. Download it from the .NET download site.
- Visual Studio 2026 (preview). Download it from the Visual Studio insiders page.

Create the sample application

Start by creating a console application that demonstrates both traditional extension methods and the new extension members syntax.

1. Create a new console application:

   .NET CLI

   ```
   dotnet new console -n PointExtensions
   cd PointExtensions
   ```

2. Copy the following code into a new file named `ExtensionMethods.cs`:

   C#

```
using System.Drawing;
using System.Numerics;

namespace ExtensionMethods;

public static class PointExtensions
{
    public static Vector2 ToVector(this Point point) =>
        new Vector2(point.X, point.Y);

    public static void Translate(this Point point, int xDist, int yDist)
    {
        point.X += xDist;
        point.Y += yDist;
    }

    public static void Scale(this Point point, int xScale, int yScale)
    {
        point.X *= xScale;
        point.Y *= yScale;
    }

    public static void Rotate(this Point point, int angleInDegrees)
    {
        double theta = ((double)angleInDegrees * Math.PI) / 180.0;
        double sinTheta = Math.Sin(theta);
        double cosTheta = Math.Cos(theta);
        double newX = (double)point.X * cosTheta - (double)point.Y *
sinTheta;
        double newY = (double)point.X * sinTheta + (double)point.Y *
cosTheta;
        point.X = (int)newX;
        point.Y = (int)newY;
    }
}
```

3. Copy the following code that demonstrates these extension methods and other uses of the System.Drawing.Point:

C#

```
using System.Drawing;
using System.Numerics;
using ExtensionMethods;

public static class ExtensionMethodsDemonstrations
{
    public static void TraditionalExtensionMethods()
    {
        OriginAsADataElement();
        ArithmeticWithPoints();
        DiscreteArithmeticWithPoints();
```

```
        ExtensionMethodsThis();
        MoreExamples();
    }

    static void OriginAsADataElement()
    {
        // Inline implementation since Point.Origin doesn't exist in
ExtensionMethods
        Point origin = Point.Empty; // Equivalent to Point.Origin
        Console.WriteLine($"Point.Origin (inline): {origin}");
        Console.WriteLine($"Same as Point.Empty: {origin == Point.Empty}");
        Console.WriteLine();
    }

    static void ArithmeticWithPoints()
    {
        Point p1 = new Point(5, 3);
        Point p2 = new Point(2, 7);

        Console.WriteLine($"Point 1: {p1}");
        Console.WriteLine($"Point 2: {p2}");

        // Inline implementation since + and - operators don't exist in
ExtensionMethods
        Point addition = new Point(p1.X + p2.X, p1.Y + p2.Y);
        Point subtraction1 = new Point(p1.X - p2.X, p1.Y - p2.Y);
        Point subtraction2 = new Point(p2.X - p1.X, p2.Y - p1.Y);

        Console.WriteLine($"Addition (p1 + p2): {addition}");
        Console.WriteLine($"Subtraction (p1 - p2): {subtraction1}");
        Console.WriteLine($"Subtraction (p2 - p1): {subtraction2}");
        Console.WriteLine();
    }

    static void DiscreteArithmeticWithPoints()
    {
        Point point = new Point(10, 8);
        int offsetX = 3;
        int offsetY = -2;
        int scaleX = 2;
        int scaleY = 3;
        int divisorX = 2;
        int divisorY = 4;

        Console.WriteLine($"Original point: {point}");
        Console.WriteLine($"Offset: ({offsetX}, {offsetY})");
        Console.WriteLine($"Scale: ({scaleX}, {scaleY})");
        Console.WriteLine($"Divisor: ({divisorX}, {divisorY})");
        Console.WriteLine();

        // Inline implementations since tuple operators don't exist in
ExtensionMethods
        Point addedOffset = new Point(point.X + offsetX, point.Y + offsetY);
        Point subtractedOffset = new Point(point.X - offsetX, point.Y -
offsetY);
```

```
        Point scaledPoint = new Point(point.X * scaleX, point.Y * scaleY);
        Point dividedPoint = new Point(point.X / divisorX, point.Y /
divisorY);

        Console.WriteLine($"point + offset: {addedOffset}");
        Console.WriteLine($"point - offset: {subtractedOffset}");
        Console.WriteLine($"point * scale: {scaledPoint}");
        Console.WriteLine($"point / divisor: {dividedPoint}");
        Console.WriteLine();
    }

    static void ExtensionMethodsThis()
    {
        // ToVector demonstration - using extension method
        Point vectorPoint = new Point(12, 16);
        Vector2 vector = vectorPoint.ToVector();
        Console.WriteLine($"Point {vectorPoint} as Vector2: {vector}");
        Console.WriteLine();

        // Translate demonstration - using extension method
        Point translatePoint = new Point(5, 5);
        Console.WriteLine($"Before Translate: {translatePoint}");
        translatePoint.Translate(3, -2);
        Console.WriteLine($"After Translate(3, -2): {translatePoint}");
        Console.WriteLine();

        // Scale demonstration - using extension method
        Point scalePoint = new Point(4, 6);
        Console.WriteLine($"Before Scale: {scalePoint}");
        scalePoint.Scale(2, 3);
        Console.WriteLine($"After Scale(2, 3): {scalePoint}");
        Console.WriteLine();

        // Rotate demonstration - using extension method
        Point rotatePoint1 = new Point(10, 0);
        Console.WriteLine($"Before Rotate: {rotatePoint1}");
        rotatePoint1.Rotate(90);
        Console.WriteLine($"After Rotate(90°): {rotatePoint1}");

        Point rotatePoint2 = new Point(5, 5);
        Console.WriteLine($"Before Rotate: {rotatePoint2}");
        rotatePoint2.Rotate(45);
        Console.WriteLine($"After Rotate(45°): {rotatePoint2}");

        Point rotatePoint3 = new Point(3, 4);
        Console.WriteLine($"Before Rotate: {rotatePoint3}");
        rotatePoint3.Rotate(180);
        Console.WriteLine($"After Rotate(180°): {rotatePoint3}");
        Console.WriteLine();
    }

    static void MoreExamples()
    {
        // Combining operators and methods
        Console.WriteLine("Scenario 1: Building a rectangle using inline
```

```
operators");
        Point topLeft = Point.Empty; // Inline equivalent of Point.Origin
        Point bottomRight = new Point(topLeft.X + 10, topLeft.Y + 8); //
Inline addition
        Point topRight = new Point(bottomRight.X, topLeft.Y);
        Point bottomLeft = new Point(topLeft.X, bottomRight.Y);

        Console.WriteLine($"Rectangle corners:");
        Console.WriteLine($"  Top-Left: {topLeft}");
        Console.WriteLine($"  Top-Right: {topRight}");
        Console.WriteLine($"  Bottom-Left: {bottomLeft}");
        Console.WriteLine($"  Bottom-Right: {bottomRight}");
        Console.WriteLine();

        // Transformation chain
        Console.WriteLine("Scenario 2: Transformation chain (mixed
methods)");
        Point transformPoint = new Point(2, 3);
        Console.WriteLine($"Starting point: {transformPoint}");

        // Scale up - using extension method
        transformPoint.Scale(3, 2);
        Console.WriteLine($"After scaling by (3, 2): {transformPoint}");

        // Translate - using inline addition
        transformPoint = new Point(transformPoint.X + 5, transformPoint.Y +
(-3));
        Console.WriteLine($"After translating by (5, -3): {transformPoint}");

        // Rotate - using extension method
        transformPoint.Rotate(45);
        Console.WriteLine($"After rotating 45�: {transformPoint}");

        // Convert to vector - using extension method
        Vector2 finalVector = transformPoint.ToVector();
        Console.WriteLine($"Final result as Vector2: {finalVector}");
        Console.WriteLine();

        // Distance calculation using inline operators and extension methods
        Console.WriteLine("Scenario 3: Distance calculation (mixed
methods)");
        Point point1 = new Point(1, 1);
        Point point2 = new Point(4, 5);
        Point difference = new Point(point2.X - point1.X, point2.Y -
point1.Y); // Inline subtraction
        Vector2 diffVector = difference.ToVector(); // Extension method
        float distance = diffVector.Length();

        Console.WriteLine($"Point 1: {point1}");
        Console.WriteLine($"Point 2: {point2}");
        Console.WriteLine($"Difference: {difference}");
        Console.WriteLine($"Distance: {distance:F2}");
        Console.WriteLine();

        Console.WriteLine("Traditional extension methods demonstration
```

```
complete!");
    }
}
```

4. Replace the content of `Program.cs` with the demonstration code:

```
C#

ExtensionMethodsDemonstrations.TraditionalExtensionMethods();
```

5. Run the sample application and examine the output.

Traditional extension methods can only add instance methods to existing types. Extension members enable you to add static properties, which provides a more natural way to extend types with constants or computed values.

Add static properties with extension members

First, examine this code in the sample:

```
C#

// Inline implementation since Point.Origin doesn't exist in ExtensionMethods
Point origin = Point.Empty; // Equivalent to Point.Origin
Console.WriteLine($"Point.Origin (inline): {origin}");
Console.WriteLine($"Same as Point.Empty: {origin == Point.Empty}");
Console.WriteLine();
```

Many apps that use 2D geometry use the concept of an `Origin`, which is the same value as Point.Empty. This code uses that fact, but some developers might create a `new Point(0,0)`, which incurs some extra work. In a given domain, you want to express these common values through static properties.

Create `NewExtensionsMembers.cs` to create extension members that solve this problem:

```
C#

using System.Drawing;
using System.Numerics;

namespace ExtensionMembers;

public static class PointExtensions
{
    extension (Point)
    {
```

```
        public static Point Origin => Point.Empty;
    }
}
```

The preceding code adds a static *extension property* to the `Point` struct. The `extension` keyword introduces an *extension block*. This extension block extends the `Point` struct.

You can use this static property as though it were a member of the `Point` struct.

C#

```
Console.WriteLine("1. Static Properties");
Console.WriteLine("--------------------");

Point origin = Point.Origin;
Console.WriteLine($"Point.Origin: {origin}");
Console.WriteLine($"Same as Point.Empty: {origin == Point.Empty}");
Console.WriteLine();
```

The `Point.Origin` property now appears as if it were part of the original `Point` type, providing a more intuitive API.

Implement arithmetic operators

Next, examine the following code that performs arithmetic with points:

C#

```
Point p1 = new Point(5, 3);
Point p2 = new Point(2, 7);

Console.WriteLine($"Point 1: {p1}");
Console.WriteLine($"Point 2: {p2}");

// Inline implementation since + and - operators don't exist in ExtensionMethods
Point addition = new Point(p1.X + p2.X, p1.Y + p2.Y);
Point subtraction1 = new Point(p1.X - p2.X, p1.Y - p2.Y);
Point subtraction2 = new Point(p2.X - p1.X, p2.Y - p1.Y);

Console.WriteLine($"Addition (p1 + p2): {addition}");
Console.WriteLine($"Subtraction (p1 - p2): {subtraction1}");
Console.WriteLine($"Subtraction (p2 - p1): {subtraction2}");
Console.WriteLine();
```

Traditional extension methods can't add operators to existing types. You must implement arithmetic operations manually, making the code verbose and harder to read. The algorithm gets duplicated whenever the operation is needed, which creates more opportunities for small

mistakes to enter the code base. It's better to place that code in one location. Add the following operators to your extension block in `NewExtensionsMembers.cs`:

```
C#

public static Point operator +(Point left, Point right) =>
    new Point(left.X + right.X, left.Y + right.Y);

public static Point operator -(Point left, Point right) =>
    new Point(left.X - right.X, left.Y - right.Y);
```

Extension members enable you to add operators directly to existing types. Now you can perform arithmetic operations using natural syntax:

```
C#

Console.WriteLine("2. Arithmetic Operators (Point + Point, Point - Point)");
Console.WriteLine("-------------------------------------------------");

Point p1 = new Point(5, 3);
Point p2 = new Point(2, 7);

Console.WriteLine($"Point 1: {p1}");
Console.WriteLine($"Point 2: {p2}");
Console.WriteLine($"Addition (p1 + p2): {p1 + p2}");
Console.WriteLine($"Subtraction (p1 - p2): {p1 - p2}");
Console.WriteLine($"Subtraction (p2 - p1): {p2 - p1}");
Console.WriteLine();
```

The extension operators make point arithmetic as natural as working with built-in numeric types.

Add more operators

You can also add extension operators for the discrete operations shown in the following code example:

```
C#

Point point = new Point(10, 8);
int offsetX = 3;
int offsetY = -2;
int scaleX = 2;
int scaleY = 3;
int divisorX = 2;
int divisorY = 4;

Console.WriteLine($"Original point: {point}");
```

```
Console.WriteLine($"Offset: ({offsetX}, {offsetY})");
Console.WriteLine($"Scale: ({scaleX}, {scaleY})");
Console.WriteLine($"Divisor: ({divisorX}, {divisorY})");
Console.WriteLine();

// Inline implementations since tuple operators don't exist in ExtensionMethods
Point addedOffset = new Point(point.X + offsetX, point.Y + offsetY);
Point subtractedOffset = new Point(point.X - offsetX, point.Y - offsetY);
Point scaledPoint = new Point(point.X * scaleX, point.Y * scaleY);
Point dividedPoint = new Point(point.X / divisorX, point.Y / divisorY);

Console.WriteLine($"point + offset: {addedOffset}");
Console.WriteLine($"point - offset: {subtractedOffset}");
Console.WriteLine($"point * scale: {scaledPoint}");
Console.WriteLine($"point / divisor: {dividedPoint}");
Console.WriteLine();
```

The `+` and `-` operators are *binary operators* and require two operands, not three. Instead of two discrete integers, use a *tuple* to specify both the `X` and `Y` deltas:

C#

```
public static Point operator *(Point left, (int dx, int dy) scale) =>
    new Point(left.X * scale.dx, left.Y * scale.dy);
public static Point operator /(Point left, (int dx, int dy) scale) =>
    new Point(left.X / scale.dx, left.Y / scale.dy);
public static Point operator +(Point left, (int dx, int dy) scale) =>
    new Point(left.X + scale.dx, left.Y + scale.dy);
public static Point operator -(Point left, (int dx, int dy) scale) =>
    new Point(left.X - scale.dx, left.Y - scale.dy);
```

The preceding operator enables elegant tuple-based operations:

C#

```
Console.WriteLine("3. Discrete Operators using tuples (Point with (int, int))");
Console.WriteLine("----------------------------------------");

Point point = new Point(10, 8);
var offset = (3, -2);
var scale = (2, 3);
var divisor = (2, 4);

Console.WriteLine($"Original point: {point}");
Console.WriteLine($"Offset tuple: {offset}");
Console.WriteLine($"Scale tuple: {scale}");
Console.WriteLine($"Divisor tuple: {divisor}");
Console.WriteLine();

Console.WriteLine($"point + offset: {point + offset}");
Console.WriteLine($"point - offset: {point - offset}");
```

```
Console.WriteLine($"point * scale: {point * scale}");
Console.WriteLine($"point / divisor: {point / divisor}");
Console.WriteLine();
```

Your extensions can include multiple overloaded operators, as long as the operands are distinct.

Migrate instance methods to extension members

Extension members also support instance methods. You don't have to change existing extension methods. The old and new forms are binary and source compatible. If you want to keep all your extensions in one container, you can. Migrating traditional extension methods to the new syntax maintains the same functionality.

Traditional extension methods

The traditional approach uses the `this` parameter syntax:

C#

```
public static Vector2 ToVector(this Point point) =>
    new Vector2(point.X, point.Y);

public static void Translate(this Point point, int xDist, int yDist)
{
    point.X += xDist;
    point.Y += yDist;
}

public static void Scale(this Point point, int xScale, int yScale)
{
    point.X *= xScale;
    point.Y *= yScale;
}

public static void Rotate(this Point point, int angleInDegress)
{
    double theta = ((double)angleInDegress * Math.PI) / 180.0;
    double sinTheta = Math.Sin(theta);
    double cosTheta = Math.Cos(theta);
    double newX = (double)point.X * cosTheta - (double)point.Y * sinTheta;
    double newY = (double)point.X * sinTheta + (double)point.Y * cosTheta;
    point.X = (int)newX;
    point.Y = (int)newY;
}
```

Extension members use a different syntax but provide the same functionality. Add the following code to your new extension members class:

C#

```
public Vector2 ToVector() =>
    new Vector2(point.X, point.Y);

public void Translate(int xDist, int yDist)
{
    point.X += xDist;
    point.Y += yDist;
}

public void Scale(int xScale, int yScale)
{
    point.X *= xScale;
    point.Y *= yScale;
}

public void Rotate(int angleInDegrees)
{
    double theta = ((double)angleInDegrees * Math.PI) / 180.0;
    double sinTheta = Math.Sin(theta);
    double cosTheta = Math.Cos(theta);
    double newX = (double)point.X * cosTheta - (double)point.Y * sinTheta;
    double newY = (double)point.X * sinTheta + (double)point.Y * cosTheta;
    point.X = (int)newX;
    point.Y = (int)newY;
}
```

The preceding code doesn't compile yet. It's the first extension you wrote that extends an *instance* of the `Point` class, instead of the type itself. To support instance extensions, your extension block needs to name the receiver parameter. Edit the following line:

C#

```
    extension (Point)
```

So that it gives a name to the `Point` instance:

C#

```
    extension (Point point)
```

Now, the code compiles. You can call these new instance methods exactly as you accessed traditional extension methods:

C#

```
Console.WriteLine("4. Instance Methods");
Console.WriteLine("------------------");

// ToVector demonstration
Point vectorPoint = new Point(12, 16);
Vector2 vector = vectorPoint.ToVector();
Console.WriteLine($"Point {vectorPoint} as Vector2: {vector}");
Console.WriteLine();

// Translate demonstration
Point translatePoint = new Point(5, 5);
Console.WriteLine($"Before Translate: {translatePoint}");
translatePoint.Translate(3, -2);
Console.WriteLine($"After Translate(3, -2): {translatePoint}");
Console.WriteLine();

// Scale demonstration
Point scalePoint = new Point(4, 6);
Console.WriteLine($"Before Scale: {scalePoint}");
scalePoint.Scale(2, 3);
Console.WriteLine($"After Scale(2, 3): {scalePoint}");
Console.WriteLine();

// Rotate demonstration
Point rotatePoint1 = new Point(10, 0);
Console.WriteLine($"Before Rotate: {rotatePoint1}");
rotatePoint1.Rotate(90);
Console.WriteLine($"After Rotate(90°): {rotatePoint1}");

Point rotatePoint2 = new Point(5, 5);
Console.WriteLine($"Before Rotate: {rotatePoint2}");
rotatePoint2.Rotate(45);
Console.WriteLine($"After Rotate(45°): {rotatePoint2}");

Point rotatePoint3 = new Point(3, 4);
Console.WriteLine($"Before Rotate: {rotatePoint3}");
rotatePoint3.Rotate(180);
Console.WriteLine($"After Rotate(180°): {rotatePoint3}");
Console.WriteLine();
```

The key difference is syntax: extension members use `extension (Type variableName)` instead of `this Type variableName`.

Completed sample

The final example shows the advantages when you combine static properties, operators, and instance methods to create comprehensive type extensions.

Compare the extension member version:

C#

```
Console.WriteLine("5. Complex Scenarios");
Console.WriteLine("--------------------");

// Combining operators and methods
Console.WriteLine("Scenario 1: Building a rectangle using operators");
Point topLeft = Point.Origin;
Point bottomRight = topLeft + (10, 8);
Point topRight = new Point(bottomRight.X, topLeft.Y);
Point bottomLeft = new Point(topLeft.X, bottomRight.Y);

Console.WriteLine($"Rectangle corners:");
Console.WriteLine($"  Top-Left: {topLeft}");
Console.WriteLine($"  Top-Right: {topRight}");
Console.WriteLine($"  Bottom-Left: {bottomLeft}");
Console.WriteLine($"  Bottom-Right: {bottomRight}");
Console.WriteLine();

// Transformation chain
Console.WriteLine("Scenario 2: Transformation chain");
Point transformPoint = new Point(2, 3);
Console.WriteLine($"Starting point: {transformPoint}");

// Scale up
transformPoint.Scale(3, 2);
Console.WriteLine($"After scaling by (3, 2): {transformPoint}");

// Translate
transformPoint = transformPoint + (5, -3);
Console.WriteLine($"After translating by (5, -3): {transformPoint}");

// Rotate
transformPoint.Rotate(45);
Console.WriteLine($"After rotating 45°: {transformPoint}");

// Convert to vector
Vector2 finalVector = transformPoint.ToVector();
Console.WriteLine($"Final result as Vector2: {finalVector}");
Console.WriteLine();

// Distance calculation using operators
Console.WriteLine("Scenario 3: Distance calculation");
Point point1 = new Point(1, 1);
Point point2 = new Point(4, 5);
Point difference = point2 - point1;
Vector2 diffVector = difference.ToVector();
float distance = diffVector.Length();

Console.WriteLine($"Point 1: {point1}");
Console.WriteLine($"Point 2: {point2}");
Console.WriteLine($"Difference: {difference}");
```

```
Console.WriteLine($"Distance: {distance:F2}");
Console.WriteLine();

Console.WriteLine("Demonstration complete!");
```

With the previous version:

C#

```
// Combining operators and methods
Console.WriteLine("Scenario 1: Building a rectangle using inline operators");
Point topLeft = Point.Empty; // Inline equivalent of Point.Origin
Point bottomRight = new Point(topLeft.X + 10, topLeft.Y + 8); // Inline addition
Point topRight = new Point(bottomRight.X, topLeft.Y);
Point bottomLeft = new Point(topLeft.X, bottomRight.Y);

Console.WriteLine($"Rectangle corners:");
Console.WriteLine($"  Top-Left: {topLeft}");
Console.WriteLine($"  Top-Right: {topRight}");
Console.WriteLine($"  Bottom-Left: {bottomLeft}");
Console.WriteLine($"  Bottom-Right: {bottomRight}");
Console.WriteLine();

// Transformation chain
Console.WriteLine("Scenario 2: Transformation chain (mixed methods)");
Point transformPoint = new Point(2, 3);
Console.WriteLine($"Starting point: {transformPoint}");

// Scale up - using extension method
transformPoint.Scale(3, 2);
Console.WriteLine($"After scaling by (3, 2): {transformPoint}");

// Translate - using inline addition
transformPoint = new Point(transformPoint.X + 5, transformPoint.Y + (-3));
Console.WriteLine($"After translating by (5, -3): {transformPoint}");

// Rotate - using extension method
transformPoint.Rotate(45);
Console.WriteLine($"After rotating 45: {transformPoint}");

// Convert to vector - using extension method
Vector2 finalVector = transformPoint.ToVector();
Console.WriteLine($"Final result as Vector2: {finalVector}");
Console.WriteLine();

// Distance calculation using inline operators and extension methods
Console.WriteLine("Scenario 3: Distance calculation (mixed methods)");
Point point1 = new Point(1, 1);
Point point2 = new Point(4, 5);
Point difference = new Point(point2.X - point1.X, point2.Y - point1.Y); // Inline
subtraction
Vector2 diffVector = difference.ToVector(); // Extension method
float distance = diffVector.Length();
```

```
Console.WriteLine($"Point 1: {point1}");
Console.WriteLine($"Point 2: {point2}");
Console.WriteLine($"Difference: {difference}");
Console.WriteLine($"Distance: {distance:F2}");
Console.WriteLine();

Console.WriteLine("Traditional extension methods demonstration complete!");
```

This example demonstrates how extension members create a cohesive API that feels like part of the original type. You can:

- Use `Point.Origin` for a meaningful starting point
- Apply mathematical operators naturally (`point + offset`, `point * scale`)
- Chain transformations using both operators and methods
- Convert between related types (`ToVector()`)

Migration benefits

When migrating from traditional extension methods to extension members, you gain:

1. **Static properties**: Add constants and computed values to types.
2. **Operators**: Enable natural mathematical and logical operations.
3. **Unified syntax**: All extension logic uses the same `extension` declaration.
4. **Type-level extensions**: Extend the type itself, not just instances.

Run the complete application to see both approaches side by side and observe how extension members provide a more integrated development experience.

Related content

- Extension methods (C# Programming Guide)
- What's new in C# 14
- Operator overloading (C# reference)

ⓘ **Note:** The author created this article with assistance from AI. Learn more

Tutorial: Create compound assignment operators

C#14 adds *user-defined compound assignment operators* that enable mutating a data structure in place, rather than creating a new instance. In previous versions of C#, the expression:

```
C#
a += b;
```

Was expanded to the following code:

```
C#
// compiler-generated code prior to C# 13:
var tmp = a + b;
a = tmp;
```

Depending on the type of `a`, this expansion leads to excessive allocations to create new instances, or copying the values of several properties to set values on the copy. Adding a user-defined operator for `+=` indicates a type can do a better job by updating the destination object in place.

C# supports the existing expansion, but it uses it only when a compound user-defined operator isn't available.

In this tutorial, you:

- ✓ Run the starting sample.
- ✓ Identify bottlenecks in the code.
- ✓ Implement new compound assignment operators.
- ✓ Analyze the completed sample.

Prerequisites

- The .NET 10 SDK. Download it from the .NET download site.
- Visual Studio 2026. Download it from the Visual Studio page.

Analyze the starting sample

Run the starter application. You can get it from the dotnet/docs GitHub repository. The sample application simulates concert attendance tracking at a theater venue. The simulation

models realistic arrival patterns throughout the evening, from early attendees to the main rush before showtime. This simulation demonstrates the object allocations when using traditional operators versus the efficiency gains possible with user-defined compound assignment operators.

The app tracks attendance through multiple theater gates (main floor and balcony sections) as concert-goers arrive. Each gate maintains a count of attendees using a `GateAttendance` record. Throughout the simulation, the code frequently updates these counts using increment (`++`) and addition (`+=`) operations. The following code shows a portion of that simulation:

C#

```
// Gate 1 - busiest entrance (target: ~100-130 people)
gates.MainFloorGates[0] += random.Next(8, 15);     // Corporate group
++gates.MainFloorGates[0];                          // Single patron
gates.MainFloorGates[0] += random.Next(20, 30);    // Tour/large group arrival
gates.MainFloorGates[0] += random.Next(5, 12);     // Family groups
++gates.MainFloorGates[0];                          // Solo attendee

// Gate 2 - second busiest (target: ~85-115 people)
gates.MainFloorGates[1] = gates.MainFloorGates[1] + random.Next(6, 12);  // Group
booking
++gates.MainFloorGates[1];                          // Single patron
gates.MainFloorGates[1] += random.Next(18, 28);    // Large family/reunion
gates.MainFloorGates[1] += random.Next(8, 15);     // Corporate/business group
gates.MainFloorGates[1] += random.Next(4, 8);      // Couples/small groups
++gates.MainFloorGates[1];                          // Individual patron
```

Identify bottlenecks in the code

With traditional operators, each operation creates a new `GateAttendance` instance, leading to significant memory allocations. The starter `GateAttendance` is *immutable*. Your code can't modify the state of the object after it's been initialized. That design decision requires copying objects if you need to modify state.

When you run the simulation, the detailed output shows:

- Gate-by-gate attendance numbers during different arrival periods.
- Total attendance tracking across all gates.
- A final comprehensive report with attendance statistics.

The following text displays some of the example output:

txt

```
Peak arrival time - all gates busy...

Peak rush period completed - all gates processed heavy traffic.

--- Gate Status After Main Rush (7:15 PM) ---
Main Floor Gates:
  Main-Floor-Gate-1: 145 attendees
  Main-Floor-Gate-2: 168 attendees
  Main-Floor-Gate-3: 149 attendees
  Main-Floor-Gate-4:  71 attendees
  Main Floor Subtotal: 533 attendees

Balcony Gates:
  Balcony-Gate-Left: 164 attendees
  Balcony-Gate-Right: 134 attendees
  Balcony Subtotal: 298 attendees

Total Current Attendance: 831 / 1000

--- Late Arrivals (7:15 PM - 7:30 PM) ---
Final patrons arriving before curtain...

Final arrivals processed - concert about to begin!
```

Examine the starter `GateAttendance` record class:

C#

```
public record class GateAttendance(string GateId)
{
    public int Count { get; init; }

    public static GateAttendance operator ++(GateAttendance gate)
    {
        GateAttendance updateGate = gate with { Count = gate.Count + 1 };
        return updateGate;
    }

    public static GateAttendance operator +(GateAttendance gate, int partySize)
    {
        GateAttendance updateGate = gate with { Count = gate.Count + partySize };
        return updateGate;
    }
}
```

The `InitialImplementation.GateAttendance` record demonstrates the traditional approach to operator overloading in C#. Notice how both the increment operator (`++`) and addition operator (`+`) create entirely new instances of `GateAttendance` using the `with` expression. Each time you write `gate++` or `gate += partySize`, the operators allocate a new record instance with the updated `Count` value, then return that new instance. While this approach maintains

immutability and thread safety, it comes at the cost of frequent memory allocations. In scenarios with many operations—like the concert simulation with hundreds of attendance updates—these allocations accumulate quickly, potentially impacting performance and increasing garbage collection pressure.

To measure this allocation behavior, try running the .NET Object Allocation tracking tool in Visual Studio. When you profile the current implementation during the concert simulation, you discover that it allocates 134 `GateAttendance` objects to complete the relatively small simulation. Each operator call creates a new instance, demonstrating how quickly allocations can accumulate in real-world scenarios. This measurement provides a concrete baseline for comparing the performance improvements you achieve with compound assignment operators.

Implement compound assignment operators

C# 14 introduces user-defined compound assignment operators that enable in-place mutations instead of creating new instances. These operators provide a more efficient alternative to the traditional pattern while maintaining the familiar compound assignment syntax.

Compound assignment operators use a new syntax that declares `void` return methods with the `operator` keyword. Add the following operators to the `GateAttendance` class:

C#

```
public void operator +=(int value) => this.property += value;
public void operator ++() => this.property++;
```

The key differences from traditional operators are:

- **Mutation**: They modify the current instance directly using `this`.
- **No new instances**: Unlike traditional operators that return new objects, compound operators modify existing ones.
- **Return type**: Compound assignment operators return `void`, not the type itself.

When the compiler encounters compound assignment expressions like `a += b` or `++a`, it follows this resolution order:

1. **Check for compound assignment operator**: If the type defines a user-defined compound assignment operator (for example, `+=` or `++`), use it directly.
2. **Fallback to traditional expansion**: If no compound operator exists, expand to the traditional form (`a = a + b`).

This means you can implement both approaches simultaneously. The compound operators take precedence when available, but the traditional operators serve as fallbacks for scenarios where compound assignment isn't suitable.

Compound assignment operators provide several advantages:

- **Reduced allocations**: Modify objects in-place instead of creating new instances.
- **Improved performance**: Eliminate temporary object creation and reduce garbage collection pressure.
- **Familiar syntax**: Use the same `+=`, `++` syntax developers already know.
- **Backward compatibility**: Traditional operators continue to work as fallbacks.

The new compound assignment operators are shown in the following code:

C#

```
public void operator ++() => Count++;

public void operator +=(int partySize) => Count += partySize;
```

> **Note**
>
> Developers familiar with C++ might wonder why only one `++` or `--` operator is required. The compiler generates the code to use either the expression before or after modification as the return value. The compiler-generated code performs the assignment using either the original value or the modified value based on whether pre-increment (`++x`) or post-increment (`x++`) was called.

Analyze finished sample

Now that you implemented the compound assignment operators, it's time to measure the performance improvement. To measure the dramatic difference in memory allocations, run the .NET Object Allocation tracking tool again on the updated code.

When you profile the application with the compound assignment operators enabled, you observe a remarkable reduction: only **10 `GateAttendance` objects** are allocated during the entire concert simulation, compared to the previous 134 allocations. This update represents a 92% reduction in object allocations!

The remaining 10 allocations come from the initial creation of the `GateAttendance` instances for each theater gate (four main floor gates + two balcony gates = six initial instances), plus a few more allocations from other parts of the simulation that don't use the compound operators.

This allocation reduction translates to real performance benefits:

- **Reduced memory pressure**: Less frequent garbage collection cycles.
- **Better cache locality**: Fewer object creations mean less memory fragmentation.
- **Improved throughput**: CPU cycles saved from allocation and collection overhead.
- **Scalability**: Benefits multiply in scenarios with higher operation volumes.

The performance improvement becomes even more significant in production applications where similar patterns occur at much larger scales—imagine tracking millions of transactions, updating thousands of counters, or processing high-frequency data streams.

Try identifying other opportunities for compound assignment operators in the codebase. Look for patterns where you use traditional assignment operations like `gates.MainFloorGates[1] = gates.MainFloorGates[1] + 4` and consider whether they could benefit from compound assignment syntax. While some of these operations are already using `+=` in the simulation code, the principle applies to any scenario where you repeatedly modify objects rather than creating new instances.

As a final experiment, change the `GateAttendance` type from a `record class` to a `record struct`. It's a different optimization, and it works in this simulation because the struct has a small memory footprint. Copying a `GateAttendance` struct isn't an expensive operation. Even so, you achieve small improvements.

Related content

- What's new in C# 14
- Operator overloading - define unary, arithmetic, equality, and comparison operators
- Analyze memory usage by using the .NET Object Allocation tool - Visual Studio

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 11/15/2025

Declare primary constructors for classes and structs

Article • 03/22/2025

C# 12 introduces primary constructors, which provide a concise syntax to declare constructors whose parameters are available anywhere in the body of the type.

This article describes how to declare a primary constructor on your type and recognize where to store primary constructor parameters. You can call primary constructors from other constructors and use primary constructor parameters in members of the type.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Understand rules for primary constructors

You can add parameters to a `struct` or `class` declaration to create a *primary constructor*. Primary constructor parameters are in scope throughout the class definition. It's important to view primary constructor parameters as *parameters* even though they are in scope throughout the class definition.

Several rules clarify that these constructors are parameters:

- Primary constructor parameters might not be stored if they aren't needed.
- Primary constructor parameters aren't members of the class. For example, a primary constructor parameter named `param` can't be accessed as `this.param`.
- Primary constructor parameters can be assigned to.
- Primary constructor parameters don't become properties, except in record types.

These rules are the same rules already defined for parameters to any method, including other constructor declarations.

Here are the most common uses for a primary constructor parameter:

- Pass as an argument to a `base()` constructor invocation
- Initialize a member field or property
- Reference the constructor parameter in an instance member

Every other constructor for a class **must** call the primary constructor, directly or indirectly, through a `this()` constructor invocation. This rule ensures that primary constructor parameters are assigned everywhere in the body of the type.

Initialize immutable properties or fields

The following code initializes two readonly (immutable) properties that are computed from primary constructor parameters:

C#

```
public readonly struct Distance(double dx, double dy)
{
    public readonly double Magnitude { get; } = Math.Sqrt(dx * dx + dy *
dy);
    public readonly double Direction { get; } = Math.Atan2(dy, dx);
}
```

This example uses a primary constructor to initialize calculated readonly properties. The field initializers for the `Magnitude` and `Direction` properties use the primary constructor parameters. The primary constructor parameters aren't used anywhere else in the struct. The code creates a struct as if it were written in the following manner:

C#

```
public readonly struct Distance
{
    public readonly double Magnitude { get; }

    public readonly double Direction { get; }

    public Distance(double dx, double dy)
    {
        Magnitude = Math.Sqrt(dx * dx + dy * dy);
        Direction = Math.Atan2(dy, dx);
    }
}
```

This feature makes it easier to use field initializers when you need arguments to initialize a field or property.

Create mutable state

The previous examples use primary constructor parameters to initialize readonly properties. You can also use primary constructors for properties that aren't readonly.

Consider the following code:

```C#
public struct Distance(double dx, double dy)
{
    public readonly double Magnitude => Math.Sqrt(dx * dx + dy * dy);
    public readonly double Direction => Math.Atan2(dy, dx);

    public void Translate(double deltaX, double deltaY)
    {
        dx += deltaX;
        dy += deltaY;
    }

    public Distance() : this(0,0) { }
}
```

In this example, the `Translate` method changes the `dx` and `dy` components, which requires the `Magnitude` and `Direction` properties be computed when accessed. The greater than or equal to (`=>`) operator designates an expression-bodied `get` accessor, whereas the equal to (`=`) operator designates an initializer.

This version of the code adds a parameterless constructor to the struct. The parameterless constructor must invoke the primary constructor, which ensures all primary constructor parameters are initialized. The primary constructor properties are accessed in a method, and the compiler creates hidden fields to represent each parameter.

The following code demonstrates an approximation of what the compiler generates. The actual field names are valid Common Intermediate Language (CIL) identifiers, but not valid C# identifiers.

```C#
public struct Distance
{
    private double __unspeakable_dx;
    private double __unspeakable_dy;

    public readonly double Magnitude => Math.Sqrt(__unspeakable_dx *
__unspeakable_dx + __unspeakable_dy * __unspeakable_dy);
    public readonly double Direction => Math.Atan2(__unspeakable_dy,
__unspeakable_dx);

    public void Translate(double deltaX, double deltaY)
    {
        __unspeakable_dx += deltaX;
        __unspeakable_dy += deltaY;
```

```
    }

    public Distance(double dx, double dy)
    {
        __unspeakable_dx = dx;
        __unspeakable_dy = dy;
    }
    public Distance() : this(0, 0) { }
}
```

Compiler-created storage

For the first example in this section, the compiler didn't need to create a field to store the value of the primary constructor parameters. However, in the second example, the primary constructor parameter is used inside a method, so the compiler must create storage for the parameters.

The compiler creates storage for any primary constructors only when the parameter is accessed in the body of a member of your type. Otherwise, the primary constructor parameters aren't stored in the object.

Use dependency injection

Another common use for primary constructors is to specify parameters for dependency injection. The following code creates a simple controller that requires a service interface for its use:

C#

```
public interface IService
{
    Distance GetDistance();
}

public class ExampleController(IService service) : ControllerBase
{
    [HttpGet]
    public ActionResult<Distance> Get()
    {
        return service.GetDistance();
    }
}
```

The primary constructor clearly indicates the parameters needed in the class. You use the primary constructor parameters as you would any other variable in the class.

Initialize base class

You can invoke the primary constructor for a base class from the primary constructor of derived class. This approach is the easiest way to write a derived class that must invoke a primary constructor in the base class. Consider a hierarchy of classes that represent different account types as a bank. The following code shows what the base class might look like:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = accountID;
    public string Owner { get; } = owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";
}
```

All bank accounts, regardless of the type, have properties for the account number and owner. In the completed application, you can add other common functionality to the base class.

Many types require more specific validation on constructor parameters. For example, the `BankAccount` class has specific requirements for the `owner` and `accountID` parameters. The `owner` parameter must not be `null` or whitespace, and the `accountID` parameter must be a string containing 10 digits. You can add this validation when you assign the corresponding properties:

C#

```
public class BankAccount(string accountID, string owner)
{
    public string AccountID { get; } = ValidAccountNumber(accountID)
        ? accountID
        : throw new ArgumentException("Invalid account number",
nameof(accountID));

    public string Owner { get; } = string.IsNullOrWhiteSpace(owner)
        ? throw new ArgumentException("Owner name cannot be empty",
nameof(owner))
        : owner;

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}";

    public static bool ValidAccountNumber(string accountID) =>
```

```
    accountID?.Length == 10 && accountID.All(c => char.IsDigit(c));
}
```

This example shows how to validate the constructor parameters before you assign them to the properties. You can use built-in methods like String.IsNullOrWhiteSpace(String) or your own validation method, such as `ValidAccountNumber`. In the example, any exceptions are thrown from the constructor, when it invokes the initializers. If a constructor parameter isn't used to assign a field, any exceptions are thrown when the constructor parameter is first accessed.

One derived class might represent a checking account:

C#

```
public class CheckingAccount(string accountID, string owner, decimal
overdraftLimit = 0) : BankAccount(accountID, owner)
{
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -overdraftLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"Account ID: {AccountID}, Owner:
{Owner}, Balance: {CurrentBalance}";
}
```

The derived `CheckingAccount` class has a primary constructor that takes all the parameters needed in the base class, and another parameter with a default value. The

primary constructor calls the base constructor with the `: BankAccount(accountID, owner)` syntax. This expression specifies both the type for the base class and the arguments for the primary constructor.

Your derived class isn't required to use a primary constructor. You can create a constructor in the derived class that invokes the primary constructor for the base class, as shown in the following example:

C#

```
public class LineOfCreditAccount : BankAccount
{
    private readonly decimal _creditLimit;
    public LineOfCreditAccount(string accountID, string owner, decimal
creditLimit) : base(accountID, owner)
    {
        _creditLimit = creditLimit;
    }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < -_creditLimit)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public override string ToString() => $"{base.ToString()}, Balance:
{CurrentBalance}";
}
```

There's one potential concern with class hierarchies and primary constructors. It's possible to create multiple copies of a primary constructor parameter because the

parameter is used in both derived and base classes. The following code creates two copies each of the `owner` and `accountID` parameters:

C#

```
public class SavingsAccount(string accountID, string owner, decimal
interestRate) : BankAccount(accountID, owner)
{
    public SavingsAccount() : this("default", "default", 0.01m) { }
    public decimal CurrentBalance { get; private set; } = 0;

    public void Deposit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "Deposit
amount must be positive");
        }
        CurrentBalance += amount;
    }

    public void Withdrawal(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount),
"Withdrawal amount must be positive");
        }
        if (CurrentBalance - amount < 0)
        {
            throw new InvalidOperationException("Insufficient funds for
withdrawal");
        }
        CurrentBalance -= amount;
    }

    public void ApplyInterest()
    {
        CurrentBalance *= 1 + interestRate;
    }

    public override string ToString() => $"Account ID: {accountID}, Owner:
{owner}, Balance: {CurrentBalance}";
}
```

The highlighted line in this example shows that the `ToString` method uses the *primary constructor parameters* (`owner` and `accountID`) rather than the *base class properties* (`Owner` and `AccountID`). The result is that the derived class, `SavingsAccount`, creates storage for the parameter copies. The copy in the derived class is different than the property in the base class. If the base class property can be modified, the instance of the

derived class doesn't see the modification. The compiler issues a warning for primary constructor parameters that are used in a derived class and passed to a base class constructor. In this instance, the fix is to use the properties of the base class.

Related content

- Parameterless constructors (C# programming guide)
- Primary constructors (Feature specification proposal)

Tutorial: Explore ideas using top-level statements to build code as you learn

Article • 01/18/2025

In this tutorial, you learn how to:

- ✓ Learn the rules governing your use of top-level statements.
- ✓ Use top-level statements to explore algorithms.
- ✓ Refactor explorations into reusable components.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Start exploring

Top-level statements enable you to avoid the extra ceremony required by placing your program's entry point in a static method in a class. The typical starting point for a new console application looks like the following code:

C#

```
using System;

namespace Application
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

The preceding code is the result of running the `dotnet new console` command and creating a new console application. Those 11 lines contain only one line of executable

code. You can simplify that program with the new top-level statements feature. That enables you to remove all but two of the lines in this program:

```
C#
```

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

> **ⓘ Important**
>
> The C# templates for .NET 6 use *top level statements*. Your application may not match the code in this article, if you've already upgraded to the .NET 6. For more information see the article on New C# templates generate top level statements
>
> The .NET 6 SDK also adds a set of *implicit* `global using` directives for projects that use the following SDKs:
>
> - Microsoft.NET.Sdk
> - Microsoft.NET.Sdk.Web
> - Microsoft.NET.Sdk.Worker
>
> These implicit `global using` directives include the most common namespaces for the project type.
>
> For more information, see the article on Implicit using directives

This feature simplifies your exploration of new ideas. You can use top-level statements for scripting scenarios, or to explore. Once you've got the basics working, you can start refactoring the code and create methods, classes, or other assemblies for reusable components you built. Top-level statements do enable quick experimentation and beginner tutorials. They also provide a smooth path from experimentation to full programs.

Top-level statements are executed in the order they appear in the file. Top-level statements can only be used in one source file in your application. The compiler generates an error if you use them in more than one file.

Build a magic .NET answer machine

For this tutorial, let's build a console application that answers a "yes" or "no" question with a random answer. You build out the functionality step by step. You can focus on

your task rather than ceremony needed for the structure of a typical program. Then, once you're happy with the functionality, you can refactor the application as you see fit.

A good starting point is to write the question back to the console. You can start by writing the following code:

C#

```
Console.WriteLine(args);
```

You don't declare an `args` variable. For the single source file that contains your top-level statements, the compiler recognizes `args` to mean the command-line arguments. The type of args is a `string[]`, as in all C# programs.

You can test your code by running the following `dotnet run` command:

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?
```

The arguments after the `--` on the command line are passed to the program. You can see the type of the `args` variable printed to the console:

Console

```
System.String[]
```

To write the question to the console, you need to enumerate the arguments and separate them with a space. Replace the `WriteLine` call with the following code:

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();
```

Now, when you run the program, it correctly displays the question as a string of arguments.

Respond with a random answer

After echoing the question, you can add the code to generate the random answer. Start by adding an array of possible answers:

C#

```
string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don’t count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];
```

This array has 10 answers that are affirmative, five that are noncommittal, and five that are negative. Next, add the following code to generate and display a random answer from the array:

C#

```
var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

You can run the application again to see the results. You should see something like the following output:

.NET CLI

```
dotnet run -- Should I use top level statements in all my programs?

Should I use top level statements in all my programs?
Better not tell you now.
```

The code to generate an answer includes a variable declaration in your top level statements. The compiler includes that declaration in the compiler generated `Main`

method. Because these variable declarations are local variables, you can't include the `static` modifier.

This code answers the questions, but let's add one more feature. You'd like your question app to simulate thinking about the answer. You can do that by adding a bit of ASCII animation, and pausing while working. Add the following code after the line that echoes the question:

C#

```
for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();
```

You also need to add a `using` directive to the top of the source file:

C#

```
using System.Threading.Tasks;
```

The `using` directives must be before any other statements in the file. Otherwise, it's a compiler error. You can run the program again and see the animation. That makes a better experience. Experiment with the length of the delay to match your taste.

The preceding code creates a set of spinning lines separated by a space. Adding the `await` keyword instructs the compiler to generate the program entry point as a method that has the `async` modifier, and returns a System.Threading.Tasks.Task. This program doesn't return a value, so the program entry point returns a `Task`. If your program returns an integer value, you would add a return statement to the end of your top-level statements. That return statement would specify the integer value to return. If your top-level statements include an `await` expression, the return type becomes System.Threading.Tasks.Task<TResult>.

Refactoring for the future

Your program should look like the following code:

C#

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

for (int i = 0; i < 20; i++)
{
    Console.Write("| -");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("/ \\");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("- |");
    await Task.Delay(50);
    Console.Write("\b\b\b");
    Console.Write("\\ /");
    await Task.Delay(50);
    Console.Write("\b\b\b");
}
Console.WriteLine();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

The preceding code is reasonable. It works. But it isn't reusable. Now that you have the application working, it's time to pull out reusable parts.

One candidate is the code that displays the waiting animation. That snippet can become a method:

You can start by creating a local function in your file. Replace the current animation with the following code:

C#

```
await ShowConsoleAnimation();

static async Task ShowConsoleAnimation()
{
    for (int i = 0; i < 20; i++)
    {
        Console.Write("| -");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("/ \\");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("- |");
        await Task.Delay(50);
        Console.Write("\b\b\b");
        Console.Write("\\ /");
        await Task.Delay(50);
        Console.Write("\b\b\b");
    }
    Console.WriteLine();
}
```

The preceding code creates a local function inside your main method. That code still isn't reusable. So, extract that code into a class. Create a new file named *utilities.cs* and add the following code:

C#

```
namespace MyNamespace
{
    public static class Utilities
    {
        public static async Task ShowConsoleAnimation()
        {
            for (int i = 0; i < 20; i++)
            {
                Console.Write("| -");
                await Task.Delay(50);
                Console.Write("\b\b\b");
```

```
                Console.Write("/ \\");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("- |");
                await Task.Delay(50);
                Console.Write("\b\b\b");
                Console.Write("\\ /");
                await Task.Delay(50);
                Console.Write("\b\b\b");
            }
            Console.WriteLine();
        }
    }
}
```

A file that has top-level statements can also contain namespaces and types at the end of the file, after the top-level statements. But for this tutorial you put the animation method in a separate file to make it more readily reusable.

Finally, you can clean the animation code to remove some duplication, by using `foreach` loop to iterate through set of animations elements defined in `animations` array.
The full `ShowConsoleAnimation` method after refactor should look similar to the following code:

C#

```
public static async Task ShowConsoleAnimation()
{
    string[] animations = ["| -", "/ \\", "- |", "\\ /"];
    for (int i = 0; i < 20; i++)
    {
        foreach (string s in animations)
        {
            Console.Write(s);
            await Task.Delay(50);
            Console.Write("\b\b\b");
        }
    }
    Console.WriteLine();
}
```

Now you have a complete application, and you refactored the reusable parts for later use. You can call the new utility method from your top-level statements, as shown in the finished version of the main program:

C#

```
using MyNamespace;
```

```
Console.WriteLine();
foreach(var s in args)
{
    Console.Write(s);
    Console.Write(' ');
}
Console.WriteLine();

await Utilities.ShowConsoleAnimation();

string[] answers =
[
    "It is certain.",       "Reply hazy, try again.",     "Don't count on
it.",
    "It is decidedly so.",  "Ask again later.",           "My reply is no.",
    "Without a doubt.",     "Better not tell you now.",   "My sources say
no.",
    "Yes – definitely.",    "Cannot predict now.",        "Outlook not so
good.",
    "You may rely on it.",  "Concentrate and ask again.", "Very doubtful.",
    "As I see it, yes.",
    "Most likely.",
    "Outlook good.",
    "Yes.",
    "Signs point to yes.",
];

var index = new Random().Next(answers.Length - 1);
Console.WriteLine(answers[index]);
```

The preceding example adds the call to `Utilities.ShowConsoleAnimation`, and adds another `using` directive.

Summary

Top-level statements make it easier to create simple programs for use to explore new algorithms. You can experiment with algorithms by trying different snippets of code. Once you learned what works, you can refactor the code to be more maintainable.

Top-level statements simplify programs that are based on console apps. These apps include Azure functions, GitHub actions, and other small utilities. For more information, see Top-level statements (C# Programming Guide).

Indices and ranges

Article • 11/14/2023

Ranges and indices provide a succinct syntax for accessing single elements or ranges in a sequence.

In this tutorial, you'll learn how to:

- ✓ Use the syntax for ranges in a sequence.
- ✓ Implicitly define a Range.
- ✓ Understand the design decisions for the start and end of each sequence.
- ✓ Learn scenarios for the Index and Range types.

Language support for indices and ranges

Indices and ranges provide a succinct syntax for accessing single elements or ranges in a sequence.

This language support relies on two new types and two new operators:

- System.Index represents an index into a sequence.
- The index from end operator `^`, which specifies that an index is relative to the end of a sequence.
- System.Range represents a sub range of a sequence.
- The range operator `..`, which specifies the start and end of a range as its operands.

Let's start with the rules for indices. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. The expression `sequence[^0]` throws an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence.Length - n`.

C#

```
private string[] words = [
                // index from start     index from end
    "first",    // 0                    ^10
    "second",   // 1                    ^9
    "third",    // 2                    ^8
    "fourth",   // 3                    ^7
    "fifth",    // 4                    ^6
    "sixth",    // 5                    ^5
    "seventh",  // 6                    ^4
    "eighth",   // 7                    ^3
    "ninth",    // 8                    ^2
```

```
    "tenth"     // 9                    ^1
];              // 10 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index. Add the following code below the initialization:

C#

```
Console.WriteLine($"The last word is < {words[^1]} >."); // The last word is
< tenth >.
```

A range specifies the *start* and *end* of a range. The start of the range is inclusive, but the end of the range is exclusive, meaning the *start* is included in the range but the *end* isn't included in the range. The range `[0..^0]` represents the entire range, just as `[0..sequence.Length]` represents the entire range.

The following code creates a subrange with the words "second", "third", and "fourth". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range.

C#

```
string[] secondThirdFourth = words[1..4]; // contains "second", "third" and
"fourth"

// < second >< third >< fourth >
foreach (var word in secondThirdFourth)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following code returns the range with "ninth" and "tenth". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` isn't included.

C#

```
 string[] lastTwo = words[^2..^0]; // contains "ninth" and "tenth"

 // < ninth >< tenth >
 foreach (var word in lastTwo)
     Console.Write($"< {word} >");
 Console.WriteLine();
```

The following examples create ranges that are open ended for the start, end, or both:

C#

```
string[] allWords = words[..]; // contains "first" through "tenth".
string[] firstPhrase = words[..4]; // contains "first" through "fourth"
string[] lastPhrase = words[6..]; // contains "seventh", "eight", "ninth"
and "tenth"

// < first >< second >< third >< fourth >< fifth >< sixth >< seventh ><
eighth >< ninth >< tenth >
foreach (var word in allWords)
    Console.Write($"< {word} >");
Console.WriteLine();

// < first >< second >< third >< fourth >
foreach (var word in firstPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();

// < seventh >< eighth >< ninth >< tenth >
foreach (var word in lastPhrase)
    Console.Write($"< {word} >");
Console.WriteLine();
```

You can also declare ranges or indices as variables. The variable can then be used inside the `[` and `]` characters:

C#

```
Index thirdFromEnd = ^3;
Console.WriteLine($"< {words[thirdFromEnd]} > "); // < eighth >
Range phrase = 1..4;
string[] text = words[phrase];

// < second >< third >< fourth >
foreach (var word in text)
    Console.Write($"< {word} >");
Console.WriteLine();
```

The following sample shows many of the reasons for those choices. Modify `x`, `y`, and `z` to try different combinations. When you experiment, use values where `x` is less than `y`, and `y` is less than `z` for valid combinations. Add the following code in a new method. Try different combinations:

C#

```
int[] numbers = [..Enumerable.Range(0, 100)];
int x = 12;
int y = 25;
int z = 36;

Console.WriteLine($"{numbers[^x]} is the same as {numbers[numbers.Length -
```

```
x]}");
Console.WriteLine($"{numbers[x..y].Length} is the same as {y - x}");

Console.WriteLine("numbers[x..y] and numbers[y..z] are consecutive and
disjoint:");
Span<int> x_y = numbers[x..y];
Span<int> y_z = numbers[y..z];
Console.WriteLine($"\tnumbers[x..y] is {x_y[0]} through {x_y[^1]},
numbers[y..z] is {y_z[0]} through {y_z[^1]}");

Console.WriteLine("numbers[x..^x] removes x elements at each end:");
Span<int> x_x = numbers[x..^x];
Console.WriteLine($"\tnumbers[x..^x] starts with {x_x[0]} and ends with
{x_x[^1]}");

Console.WriteLine("numbers[..x] means numbers[0..x] and numbers[x..] means
numbers[x..^0]");
Span<int> start_x = numbers[..x];
Span<int> zero_x = numbers[0..x];
Console.WriteLine($"\t{start_x[0]}..{start_x[^1]} is the same as
{zero_x[0]}..{zero_x[^1]}");
Span<int> z_end = numbers[z..];
Span<int> z_zero = numbers[z..^0];
Console.WriteLine($"\t{z_end[0]}..{z_end[^1]} is the same as {z_zero[0]}..
{z_zero[^1]}");
```

Not only arrays support indices and ranges. You can also use indices and ranges with string, Span<T>, or ReadOnlySpan<T>.

Implicit range operator expression conversions

When using the range operator expression syntax, the compiler implicitly converts the start and end values to an Index and from them, creates a new Range instance. The following code shows an example implicit conversion from the range operator expression syntax, and its corresponding explicit alternative:

C#

```
Range implicitRange = 3..^5;

Range explicitRange = new(
    start: new Index(value: 3, fromEnd: false),
    end: new Index(value: 5, fromEnd: true));

if (implicitRange.Equals(explicitRange))
{
    Console.WriteLine(
        $"The implicit range '{implicitRange}' equals the explicit range
'{explicitRange}'");
}
```

```
// Sample output:
//     The implicit range '3..^5' equals the explicit range '3..^5'
```

> **Important**
>
> Implicit conversions from Int32 to Index throw an ArgumentOutOfRangeException when the value is negative. Likewise, the `Index` constructor throws an `ArgumentOutOfRangeException` when the `value` parameter is negative.

Type support for indices and ranges

Indexes and ranges provide clear, concise syntax to access a single element or a range of elements in a sequence. An index expression typically returns the type of the elements of a sequence. A range expression typically returns the same sequence type as the source sequence.

Any type that provides an indexer with an Index or Range parameter explicitly supports indices or ranges respectively. An indexer that takes a single Range parameter may return a different sequence type, such as System.Span<T>.

> **Important**
>
> The performance of code using the range operator depends on the type of the sequence operand.
>
> The time complexity of the range operator depends on the sequence type. For example, if the sequence is a `string` or an array, then the result is a copy of the specified section of the input, so the time complexity is *O(N)* (where N is the length of the range). On the other hand, if it's a System.Span<T> or a System.Memory<T>, the result references the same backing store, which means there is no copy and the operation is *O(1)*.
>
> In addition to the time complexity, this causes extra allocations and copies, impacting performance. In performance sensitive code, consider using `Span<T>` or `Memory<T>` as the sequence type, since the range operator does not allocate for them.

A type is **countable** if it has a property named `Length` or `Count` with an accessible getter and a return type of `int`. A countable type that doesn't explicitly support indices or ranges may provide an implicit support for them. For more information, see the Implicit

Index support and Implicit Range support sections of the feature proposal note. Ranges using implicit range support return the same sequence type as the source sequence.

For example, the following .NET types support both indices and ranges: String, Span<T>, and ReadOnlySpan<T>. The List<T> supports indices but doesn't support ranges.

Array has more nuanced behavior. Single dimension arrays support both indices and ranges. Multi-dimensional arrays don't support indexers or ranges. The indexer for a multi-dimensional array has multiple parameters, not a single parameter. Jagged arrays, also referred to as an array of arrays, support both ranges and indexers. The following example shows how to iterate a rectangular subsection of a jagged array. It iterates the section in the center, excluding the first and last three rows, and the first and last two columns from each selected row:

C#

```
int[][] jagged =
[
   [0, 1, 2, 3, 4, 5, 6, 7, 8, 9],
   [10,11,12,13,14,15,16,17,18,19],
   [20,21,22,23,24,25,26,27,28,29],
   [30,31,32,33,34,35,36,37,38,39],
   [40,41,42,43,44,45,46,47,48,49],
   [50,51,52,53,54,55,56,57,58,59],
   [60,61,62,63,64,65,66,67,68,69],
   [70,71,72,73,74,75,76,77,78,79],
   [80,81,82,83,84,85,86,87,88,89],
   [90,91,92,93,94,95,96,97,98,99],
];

var selectedRows = jagged[3..^3];

foreach (var row in selectedRows)
{
    var selectedColumns = row[2..^2];
    foreach (var cell in selectedColumns)
    {
        Console.Write($"{cell}, ");
    }
    Console.WriteLine();
}
```

In all cases, the range operator for Array allocates an array to store the elements returned.

Scenarios for indices and ranges

You'll often use ranges and indices when you want to analyze a portion of a larger sequence. The new syntax is clearer in reading exactly what portion of the sequence is involved. The local function `MovingAverage` takes a Range as its argument. The method then enumerates just that range when calculating the min, max, and average. Try the following code in your project:

C#

```
int[] sequence = Sequence(1000);

for(int start = 0; start < sequence.Length; start += 100)
{
    Range r = start..(start+10);
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:    \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

for (int start = 0; start < sequence.Length; start += 100)
{
    Range r = ^(start + 10)..^start;
    var (min, max, average) = MovingAverage(sequence, r);
    Console.WriteLine($"From {r.Start} to {r.End}:  \tMin: {min},\tMax:
{max},\tAverage: {average}");
}

(int min, int max, double average) MovingAverage(int[] subSequence, Range
range) =>
    (
        subSequence[range].Min(),
        subSequence[range].Max(),
        subSequence[range].Average()
    );

int[] Sequence(int count) => [..Enumerable.Range(0, count).Select(x => (int)
(Math.Sqrt(x) * 100))];
```

A Note on Range Indices and Arrays

When taking a range from an array, the result is an array that is copied from the initial array, rather than referenced. Modifying values in the resulting array will not change values in the initial array.

For example:

C#

```
var arrayOfFiveItems = new[] { 1, 2, 3, 4, 5 };

var firstThreeItems = arrayOfFiveItems[..3]; // contains 1,2,3
firstThreeItems[0] =  11; // now contains 11,2,3

Console.WriteLine(string.Join(",", firstThreeItems));
Console.WriteLine(string.Join(",", arrayOfFiveItems));

// output:
// 11,2,3
// 1,2,3,4,5
```

See also

- Member access operators and expressions

Tutorial: Express your design intent more clearly with nullable and non-nullable reference types

Nullable reference types complement reference types the same way nullable value types complement value types. You declare a variable to be a **nullable reference type** by appending a `?` to the type. For example, `string?` represents a nullable `string`. You can use these new types to more clearly express your design intent: some variables *must always have a value* while others *might be missing a value*.

In this tutorial, you learn how to:

- ✓ Incorporate nullable and non-nullable reference types into your designs
- ✓ Enable nullable reference type checks throughout your code.
- ✓ Write code where the compiler enforces those design decisions.
- ✓ Use the nullable reference feature in your own designs

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Incorporate nullable reference types into your designs

In this tutorial, you'll build a library that models running a survey. The code uses both nullable reference types and non-nullable reference types to represent the real-world concepts. The survey questions can never be null. A respondent might prefer not to answer a question. The responses might be `null` in this case.

The code you write for this sample expresses that intent, and the compiler enforces that intent.

Create the application and enable nullable reference types

Create a new console application either in Visual Studio or from the command line using `dotnet new console`. Name the application `NullableIntroduction`. Once you create the application, you need to specify that the entire project compiles in an enabled **nullable annotation context**. Open the *.csproj* file and add a `Nullable` element to the `PropertyGroup` element. Set its value to `enable`. You must opt in to the **nullable reference types** feature in projects created before C# 11 / .NET 7. Once the feature is turned on, existing reference variable declarations become **non-nullable reference types**. While that decision helps find issues where existing code might not have proper null-checks, it might not accurately reflect your original design intent:

XML

```
<Nullable>enable</Nullable>
```

Design the types for the application

This survey application requires creating these classes:

- A class that models the list of questions.
- A class that models a list of people contacted for the survey.
- A class that models the answers from a person that took the survey.

These types make use of both nullable and non-nullable reference types to express which members are required and which members are optional. Nullable reference types communicate that design intent clearly:

- The questions that are part of the survey can never be null: It makes no sense to ask an empty question.
- The respondents can never be null. You want to track people you contacted, even respondents that declined to participate.
- Any response to a question might be null. Respondents can decline to answer some or all questions.

You might be so accustomed to reference types that allow `null` values that you might miss other opportunities to declare non-nullable instances:

- The collection of questions should be non-nullable.
- The collection of respondents should be non-nullable.

As you write the code, you see that a non-nullable reference type as the default for references avoids common mistakes that could lead to NullReferenceExceptions. One lesson from this tutorial is that you made decisions about which variables could or couldn't be `null`. The language didn't provide syntax to express those decisions. Now it does.

The app you build does the following steps:

1. Creates a survey and adds questions to it.
2. Creates a pseudo-random set of respondents for the survey.
3. Contacts respondents until the completed survey size reaches the goal number.
4. Writes out important statistics on the survey responses.

Build the survey with nullable and non-nullable reference types

The first code you write creates the survey. You write classes to model a survey question and a survey run. Your survey has three types of questions, distinguished by the format of the answer: Yes/No answers, number answers, and text answers. Create a `public SurveyQuestion` class:

C#

```
namespace NullableIntroduction
{
    public class SurveyQuestion
    {
    }
}
```

The compiler interprets every reference type variable declaration as a **non-nullable** reference type for code in an enabled nullable annotation context. You can see your first warning by adding properties for the question text and the type of question, as shown in the following code:

C#

```
namespace NullableIntroduction
{
    public enum QuestionType
    {
        YesNo,
        Number,
        Text
    }

    public class SurveyQuestion
    {
        public string QuestionText { get; }
        public QuestionType TypeOfQuestion { get; }
    }
}
```

Because you didn't initialize `QuestionText`, the compiler issues a warning that a non-nullable property wasn't initialized. Your design requires the question text to be non-null, so you add a constructor to initialize it and the `QuestionType` value as well. The finished class definition looks like the following code:

C#

```
namespace NullableIntroduction;

public enum QuestionType
{
    YesNo,
    Number,
    Text
}

public class SurveyQuestion
{
    public string QuestionText { get; }
    public QuestionType TypeOfQuestion { get; }

    public SurveyQuestion(QuestionType typeOfQuestion, string text) =>
        (TypeOfQuestion, QuestionText) = (typeOfQuestion, text);
}
```

Adding the constructor removes the warning. The constructor argument is also a non-nullable reference type, so the compiler doesn't issue any warnings.

Next, create a `public` class named `SurveyRun`. This class contains a list of `SurveyQuestion` objects and methods to add questions to the survey, as shown in the following code:

C#

```
using System.Collections.Generic;

namespace NullableIntroduction
{
    public class SurveyRun
    {
        private List<SurveyQuestion> surveyQuestions = new List<SurveyQuestion>();

        public void AddQuestion(QuestionType type, string question) =>
            AddQuestion(new SurveyQuestion(type, question));
        public void AddQuestion(SurveyQuestion surveyQuestion) =>
surveyQuestions.Add(surveyQuestion);
    }
}
```

As before, you must initialize the list object to a non-null value or the compiler issues a warning. There are no null checks in the second overload of `AddQuestion` because the compiler helps enforce the non-nullable contract: You declared that variable to be non-nullable. While the compiler warns about potential null assignments, runtime null values are still possible. For public APIs, consider adding argument validation even for non-nullable reference types, since client code might not have nullable reference types enabled or could intentionally pass null.

Switch to *Program.cs* in your editor and replace the contents of `Main` with the following lines of code:

C#

```
var surveyRun = new SurveyRun();
surveyRun.AddQuestion(QuestionType.YesNo, "Has your code ever thrown a
NullReferenceException?");
surveyRun.AddQuestion(new SurveyQuestion(QuestionType.Number, "How many times (to
the nearest 100) has that happened?"));
surveyRun.AddQuestion(QuestionType.Text, "What is your favorite color?");
```

Because the entire project is in an enabled nullable annotation context, you get warnings when you pass `null` to any method expecting a non-nullable reference type. Try it by adding the following line to `Main`:

C#

```
surveyRun.AddQuestion(QuestionType.Text, default);
```

Create respondents and get answers to the survey

Next, write the code that generates answers to the survey. This process involves several small tasks:

1. Build a method that generates respondent objects. These objects represent people asked to fill out the survey.
2. Build logic to simulate asking the questions to a respondent and collecting answers or noting that a respondent didn't answer.
3. Repeat until enough respondents answer the survey.

You need a class to represent a survey response, so add that now. Enable nullable support. Add an `Id` property and a constructor that initializes it, as shown in the following code:

C#

```
namespace NullableIntroduction
{
    public class SurveyResponse
    {
        public int Id { get; }

        public SurveyResponse(int id) => Id = id;
    }
}
```

Next, add a `static` method to create new participants by generating a random ID:

C#

```
private static readonly Random randomGenerator = new Random();
public static SurveyResponse GetRandomId() => new
SurveyResponse(randomGenerator.Next());
```

The main responsibility of this class is to generate the responses for a participant to the questions in the survey. This responsibility has a few steps:

1. Ask for participation in the survey. If the person doesn't consent, return a missing (or null) response.
2. Ask each question and record the answer. Each answer might also be missing (or null).

Add the following code to your `SurveyResponse` class:

C#

```
private Dictionary<int, string>? surveyResponses;
public bool AnswerSurvey(IEnumerable<SurveyQuestion> questions)
{
    if (ConsentToSurvey())
    {
        surveyResponses = new Dictionary<int, string>();
        int index = 0;
        foreach (var question in questions)
        {
            var answer = GenerateAnswer(question);
            if (answer != null)
            {
                surveyResponses.Add(index, answer);
            }
            index++;
        }
    }
    return surveyResponses != null;
}

private bool ConsentToSurvey() => randomGenerator.Next(0, 2) == 1;
```

```
private string? GenerateAnswer(SurveyQuestion question)
{
    switch (question.TypeOfQuestion)
    {
        case QuestionType.YesNo:
            int n = randomGenerator.Next(-1, 2);
            return (n == -1) ? default : (n == 0) ? "No" : "Yes";
        case QuestionType.Number:
            n = randomGenerator.Next(-30, 101);
            return (n < 0) ? default : n.ToString();
        case QuestionType.Text:
        default:
            switch (randomGenerator.Next(0, 5))
            {
                case 0:
                    return default;
                case 1:
                    return "Red";
                case 2:
                    return "Green";
                case 3:
                    return "Blue";
            }
            return "Red. No, Green. Wait.. Blue... AAARGGGGGHHH!";
    }
}
```

The storage for the survey answers is a `Dictionary<int, string>?`, indicating that it might be null. You're using the new language feature to declare your design intent, both to the compiler and to anyone reading your code later. If you ever dereference `surveyResponses` without checking for the `null` value first, you get a compiler warning. You don't get a warning in the `AnswerSurvey` method because the compiler can determine the `surveyResponses` variable was set to a non-null value in the preceding code.

Using `null` for missing answers highlights a key point for working with nullable reference types: your goal isn't to remove all `null` values from your program. Rather, your goal is to ensure that the code you write expresses the intent of your design. Missing values are a necessary concept to express in your code. The `null` value is a clear way to express those missing values. Trying to remove all `null` values leads to defining some other way to express those missing values without `null`.

Next, you need to write the `PerformSurvey` method in the `SurveyRun` class. Add the following code in the `SurveyRun` class:

C#

```
private List<SurveyResponse>? respondents;
public void PerformSurvey(int numberOfRespondents)
{
    int respondentsConsenting = 0;
    respondents = new List<SurveyResponse>();
    while (respondentsConsenting < numberOfRespondents)
    {
        var respondent = SurveyResponse.GetRandomId();
        if (respondent.AnswerSurvey(surveyQuestions))
            respondentsConsenting++;
        respondents.Add(respondent);
    }
}
```

Here again, your choice of a nullable `List<SurveyResponse>?` indicates the response might be null. That indicates the survey hasn't been given to any respondents yet. Notice that respondents are added until enough consent.

The last step to run the survey is to add a call to perform the survey at the end of the `Main` method:

C#

```
surveyRun.PerformSurvey(50);
```

Examine survey responses

The last step is to display survey results. You add code to many of the classes you wrote. This code demonstrates the value of distinguishing nullable and non-nullable reference types. Start by adding the following two expression-bodied members to the `SurveyResponse` class:

C#

```
public bool AnsweredSurvey => surveyResponses != null;
public string Answer(int index) => surveyResponses?.GetValueOrDefault(index) ?? "No
answer";
```

Because `surveyResponses` is a nullable reference type, null checks are necessary before de-referencing it. The `Answer` method returns a non-nullable string, so we have to cover the case of a missing answer by using the null-coalescing operator.

Next, add these three expression-bodied members to the `SurveyRun` class:

C#

```
public IEnumerable<SurveyResponse> AllParticipants => (respondents ??
Enumerable.Empty<SurveyResponse>());
public ICollection<SurveyQuestion> Questions => surveyQuestions;
public SurveyQuestion GetQuestion(int index) => surveyQuestions[index];
```

The `AllParticipants` member must take into account that the `respondents` variable might be null, but the return value can't be null. If you change that expression by removing the `??` and the empty sequence that follows, the compiler warns you the method might return `null` and its return signature returns a non-nullable type.

Finally, add the following loop at the bottom of the `Main` method:

C#

```
foreach (var participant in surveyRun.AllParticipants)
{
    Console.WriteLine($"Participant: {participant.Id}:");
    if (participant.AnsweredSurvey)
    {
        for (int i = 0; i < surveyRun.Questions.Count; i++)
        {
            var answer = participant.Answer(i);
            Console.WriteLine($"\t{surveyRun.GetQuestion(i).QuestionText} :
{answer}");
        }
    }
    else
    {
        Console.WriteLine("\tNo responses");
    }
}
```

You don't need any `null` checks in this code because you designed the underlying interfaces so that they all return non-nullable reference types. The compiler's static analysis helps ensure these design contracts are followed.

Get the code

You can get the code for the finished tutorial from our samples repository in the csharp/NullableIntroduction folder.

Experiment by changing the type declarations between nullable and non-nullable reference types. See how that generates different warnings to ensure you don't accidentally dereference a `null`.

Next steps

Learn how to use nullable reference type when using Entity Framework:

Entity Framework Core Fundamentals: Working with Nullable Reference Types

---

Last updated on 11/20/2025

Use pattern matching to build your class behavior for better code

Article • 12/04/2024

The pattern matching features in C# provide syntax to express your algorithms. You can use these techniques to implement the behavior in your classes. You can combine object-oriented class design with a data-oriented implementation to provide concise code while modeling real-world objects.

In this tutorial, you learn how to:

- ✓ Express your object oriented classes using data patterns.
- ✓ Implement those patterns using C#'s pattern matching features.
- ✓ Leverage compiler diagnostics to validate your implementation.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Build a simulation of a canal lock

In this tutorial, you build a C# class that simulates a canal lock. Briefly, a canal lock is a device that raises and lowers boats as they travel between two stretches of water at different levels. A lock has two gates and some mechanism to change the water level.

In its normal operation, a boat enters one of the gates while the water level in the lock matches the water level on the side the boat enters. Once in the lock, the water level is changed to match the water level where the boat leaves the lock. Once the water level matches that side, the gate on the exit side opens. Safety measures make sure an operator can't create a dangerous situation in the canal. The water level can be changed only when both gates are closed. At most one gate can be open. To open a gate, the water level in the lock must match the water level outside the gate being opened.

You can build a C# class to model this behavior. A `CanalLock` class would support commands to open or close either gate. It would have other commands to raise or lower the water. The class should also support properties to read the current state of both gates and the water level. Your methods implement the safety measures.

Define a class

You build a console application to test your `CanalLock` class. Create a new console project for .NET 5 using either Visual Studio or the .NET CLI. Then, add a new class and name it `CanalLock`. Next, design your public API, but leave the methods not implemented:

C#

```
public enum WaterLevel
{
    Low,
    High
}
public class CanalLock
{
    // Query canal lock state:
    public WaterLevel CanalLockWaterLevel { get; private set; } =
WaterLevel.Low;
    public bool HighWaterGateOpen { get; private set; } = false;
    public bool LowWaterGateOpen { get; private set; } = false;

    // Change the upper gate.
    public void SetHighGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change the lower gate.
    public void SetLowGate(bool open)
    {
        throw new NotImplementedException();
    }

    // Change water level.
    public void SetWaterLevel(WaterLevel newLevel)
    {
        throw new NotImplementedException();
    }

    public override string ToString() =>
        $"The lower gate is {(LowWaterGateOpen ? "Open" : "Closed")}. " +
        $"The upper gate is {(HighWaterGateOpen ? "Open" : "Closed")}. " +
        $"The water level is {CanalLockWaterLevel}.";
}
```

The preceding code initializes the object so both gates are closed, and the water level is low. Next, write the following test code in your `Main` method to guide you as you create a first implementation of the class:

C#

```
// Create a new canal lock:
var canalGate = new CanalLock();

// State should be doors closed, water level low:
Console.WriteLine(canalGate);

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat enters lock from lower gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");

canalGate.SetWaterLevel(WaterLevel.High);
Console.WriteLine($"Raise the water level: {canalGate}");

canalGate.SetHighGate(open: true);
Console.WriteLine($"Open the higher gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");
Console.WriteLine("Boat enters lock from upper gate");

canalGate.SetHighGate(open: false);
Console.WriteLine($"Close the higher gate: {canalGate}");

canalGate.SetWaterLevel(WaterLevel.Low);
Console.WriteLine($"Lower the water level: {canalGate}");

canalGate.SetLowGate(open: true);
Console.WriteLine($"Open the lower gate:  {canalGate}");

Console.WriteLine("Boat exits lock at upper gate");

canalGate.SetLowGate(open: false);
Console.WriteLine($"Close the lower gate:  {canalGate}");
```

Next, add a first implementation of each method in the `CanalLock` class. The following code implements the methods of the class without concern to the safety rules. You add safety tests later:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = open;
}

// Change the lower gate.
```

```
public void SetLowGate(bool open)
{
    LowWaterGateOpen = open;
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = newLevel;
}
```

The tests you wrote so far pass. You implemented the basics. Now, write a test for the first failure condition. At the end of the previous tests, both gates are closed, and the water level is set to low. Add a test to try opening the upper gate:

C#

```
Console.WriteLine("=============================================");
Console.WriteLine("     Test invalid commands");
// Open "wrong" gate (2 tests)
try
{
    canalGate = new CanalLock();
    canalGate.SetHighGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation: Can't open the high gate. Water is
low.");
}
Console.WriteLine($"Try to open upper gate: {canalGate}");
```

This test fails because the gate opens. As a first implementation, you could fix it with the following code:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    if (open && (CanalLockWaterLevel == WaterLevel.High))
        HighWaterGateOpen = true;
    else if (open && (CanalLockWaterLevel == WaterLevel.Low))
        throw new InvalidOperationException("Cannot open high gate when the
water is low");
}
```

Your tests pass. But, as you add more tests, you add more `if` clauses and test different properties. Soon, these methods get too complicated as you add more conditionals.

Implement the commands with patterns

A better way is to use *patterns* to determine if the object is in a valid state to execute a command. You can express if a command is allowed as a function of three variables: the state of the gate, the level of the water, and the new setting:

Expand table

New setting	Gate state	Water Level	Result
Closed	Closed	High	Closed
Closed	Closed	Low	Closed
Closed	Open	High	Closed
~~Closed~~	~~Open~~	~~Low~~	~~Closed~~
Open	Closed	High	Open
Open	Closed	Low	Closed (Error)
Open	Open	High	Open
~~Open~~	~~Open~~	~~Low~~	~~Closed (Error)~~

The fourth and last rows in the table have strike through text because they're invalid. The code you're adding now should make sure the high water gate is never opened when the water is low. Those states can be coded as a single switch expression (remember that `false` indicates "Closed"):

C#

```
HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel) switch
{
    (false, false, WaterLevel.High) => false,
    (false, false, WaterLevel.Low) => false,
    (false, true, WaterLevel.High) => false,
    (false, true, WaterLevel.Low) => false, // should never happen
    (true, false, WaterLevel.High) => true,
    (true, false, WaterLevel.Low) => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
    (true, true, WaterLevel.High) => true,
    (true, true, WaterLevel.Low) => false, // should never happen
};
```

Try this version. Your tests pass, validating the code. The full table shows the possible combinations of inputs and results. That means you and other developers can quickly

look at the table and see that you covered all the possible inputs. Even easier, the compiler can help as well. After you add the previous code, you can see that the compiler generates a warning: *CS8524* indicates the switch expression doesn't cover all possible inputs. The reason for that warning is that one of the inputs is an `enum` type. The compiler interprets "all possible inputs" as all inputs from the underlying type, typically an `int`. This `switch` expression only checks the values declared in the `enum`. To remove the warning, you can add a catch-all discard pattern for the last arm of the expression. This condition throws an exception, because it indicates invalid input:

```
C#
```

```
_  => throw new InvalidOperationException("Invalid internal state"),
```

The preceding switch arm must be last in your `switch` expression because it matches all inputs. Experiment by moving it earlier in the order. That causes a compiler error *CS8510* for unreachable code in a pattern. The natural structure of switch expressions enables the compiler to generate errors and warnings for possible mistakes. The compiler "safety net" makes it easier for you to create correct code in fewer iterations, and the freedom to combine switch arms with wildcards. The compiler issues errors if your combination results in unreachable arms you didn't expect, and warnings if you remove a needed arm.

The first change is to combine all the arms where the command is to close the gate; that's always allowed. Add the following code as the first arm in your switch expression:

```
C#
```

```
(false, _, _) => false,
```

After you add the previous switch arm, you'll get four compiler errors, one on each of the arms where the command is `false`. Those arms are already covered by the newly added arm. You can safely remove those four lines. You intended this new switch arm to replace those conditions.

Next, you can simplify the four arms where the command is to open the gate. In both cases where the water level is high, the gate can be opened. (In one, it's already open.) One case where the water level is low throws an exception, and the other shouldn't happen. It should be safe to throw the same exception if the water lock is already in an invalid state. You can make the following simplifications for those arms:

```
C#
```

```
(true, _, WaterLevel.High) => true,
(true, false, WaterLevel.Low) => throw new InvalidOperationException("Cannot
open high gate when the water is low"),
_ => throw new InvalidOperationException("Invalid internal state"),
```

Run your tests again, and they pass. Here's the final version of the `SetHighGate` method:

C#

```
// Change the upper gate.
public void SetHighGate(bool open)
{
    HighWaterGateOpen = (open, HighWaterGateOpen, CanalLockWaterLevel)
switch
    {
        (false, _,     _)               => false,
        (true, _,      WaterLevel.High) => true,
        (true, false, WaterLevel.Low)   => throw new
InvalidOperationException("Cannot open high gate when the water is low"),
        _                               => throw new
InvalidOperationException("Invalid internal state"),
    };
}
```

Implement patterns yourself

Now that you've seen the technique, fill in the `SetLowGate` and `SetWaterLevel` methods yourself. Start by adding the following code to test invalid operations on those methods:

C#

```
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetLowGate(open: true);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't open the lower gate. Water
is high.");
}
Console.WriteLine($"Try to open lower gate: {canalGate}");
// change water level with gate open (2 tests)
Console.WriteLine();
Console.WriteLine();
try
```

```
{
    canalGate = new CanalLock();
    canalGate.SetLowGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.High);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't raise water when the lower
gate is open.");
}
Console.WriteLine($"Try to raise water with lower gate open: {canalGate}");
Console.WriteLine();
Console.WriteLine();
try
{
    canalGate = new CanalLock();
    canalGate.SetWaterLevel(WaterLevel.High);
    canalGate.SetHighGate(open: true);
    canalGate.SetWaterLevel(WaterLevel.Low);
}
catch (InvalidOperationException)
{
    Console.WriteLine("invalid operation: Can't lower water when the high
gate is open.");
}
Console.WriteLine($"Try to lower water with high gate open: {canalGate}");
```

Run your application again. You can see the new tests fail, and the canal lock gets into an invalid state. Try to implement the remaining methods yourself. The method to set the lower gate should be similar to the method to set the upper gate. The method that changes the water level has different checks, but should follow a similar structure. You might find it helpful to use the same process for the method that sets the water level. Start with all four inputs: The state of both gates, the current state of the water level, and the requested new water level. The switch expression should start with:

C#

```
CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
{
    // elided
};
```

You have 16 total switch arms to fill in. Then, test and simplify.

Did you make methods something like this?

C#

```
// Change the lower gate.
public void SetLowGate(bool open)
{
    LowWaterGateOpen = (open, LowWaterGateOpen, CanalLockWaterLevel) switch
    {
        (false, _, _) => false,
        (true, _, WaterLevel.Low) => true,
        (true, false, WaterLevel.High) => throw new
InvalidOperationException("Cannot open low gate when the water is high"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}

// Change water level.
public void SetWaterLevel(WaterLevel newLevel)
{
    CanalLockWaterLevel = (newLevel, CanalLockWaterLevel, LowWaterGateOpen,
HighWaterGateOpen) switch
    {
        (WaterLevel.Low, WaterLevel.Low, true, false) => WaterLevel.Low,
        (WaterLevel.High, WaterLevel.High, false, true) => WaterLevel.High,
        (WaterLevel.Low, _, false, false) => WaterLevel.Low,
        (WaterLevel.High, _, false, false) => WaterLevel.High,
        (WaterLevel.Low, WaterLevel.High, false, true) => throw new
InvalidOperationException("Cannot lower water when the high gate is open"),
        (WaterLevel.High, WaterLevel.Low, true, false) => throw new
InvalidOperationException("Cannot raise water when the low gate is open"),
        _ => throw new InvalidOperationException("Invalid internal state"),
    };
}
```

Your tests should pass, and the canal lock should operate safely.

Summary

In this tutorial, you learned to use pattern matching to check the internal state of an object before applying any changes to that state. You can check combinations of properties. Once you built tables for any of those transitions, you test your code, then simplify for readability and maintainability. These initial refactorings might suggest further refactorings that validate internal state or manage other API changes. This tutorial combined classes and objects with a more data-oriented, pattern-based approach to implement those classes.

String interpolation in C#

This tutorial shows you how to use string interpolation to format and include expression results in a result string. The examples assume that you're familiar with basic C# concepts and .NET type formatting. For more information about formatting types in .NET, see Formatting types in .NET.

Introduction

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You can embed any valid C# expression that returns a value in an interpolated string. In the following example, as soon as an expression is evaluated, its result is converted into a string and included in a result string:

C#

```
double a = 3;
double b = 4;
Console.WriteLine($"Area of the right triangle with legs of {a} and {b} is {0.5 * a * b}");
Console.WriteLine($"Length of the hypotenuse of the right triangle with legs of {a} and {b} is {CalculateHypotenuse(a, b)}");
double CalculateHypotenuse(double leg1, double leg2) => Math.Sqrt(leg1 * leg1 + leg2 * leg2);
// Output:
// Area of the right triangle with legs of 3 and 4 is 6
// Length of the hypotenuse of the right triangle with legs of 3 and 4 is 5
```

As the example shows, you include an expression in an interpolated string by enclosing it with braces:

C#

```
{<interpolationExpression>}
```

Interpolated strings support all the capabilities of the string composite formatting feature. That makes them a more readable alternative to the use of the String.Format method. Every interpolated string must have:

- A string literal that begins with the `$` character before its opening quotation mark character. There can't be any spaces between the `$` symbol and the quotation mark character.
- One or more *interpolation expressions*. You indicate an interpolation expression with an opening and closing brace (`{` and `}`). You can put any C# expression that returns a value

(including `null`) inside the braces.

C# evaluates the expression between the `{` and `}` characters with the following rules:

- If the interpolation expression evaluates to `null`, an empty string ("", or String.Empty) is used.
- If the interpolation expression doesn't evaluate to `null`, typically the `ToString` method of the result type is called.

How to specify a format string for an interpolation expression

To specify a format string supported by the type of the expression result, follow the interpolation expression with a colon (":") and the format string:

C#

```
{<interpolationExpression>:<formatString>}
```

The following example shows how to specify standard and custom format strings for expressions that produce date and time or numeric results:

C#

```
var date = new DateTime(1731, 11, 25);
Console.WriteLine($"On {date:dddd, MMMM dd, yyyy} L. Euler introduced the letter e
to denote {Math.E:F5}.");
// Output:
// On Sunday, November 25, 1731 L. Euler introduced the letter e to denote 2.71828.
```

For more information, see the Format string component section of the Composite formatting article.

How to control the field width and alignment of the formatted interpolation expression

To specify the minimum field width and the alignment of the formatted expression result, follow the interpolation expression with a comma (",") and the constant expression:

C#

```
{<interpolationExpression>,<width>}
```

The following code sample uses the minimum field width to create a tabular output:

C#

```
var titles = new Dictionary<string, string>()
{
    ["Doyle, Arthur Conan"] = "Hound of the Baskervilles, The",
    ["London, Jack"] = "Call of the Wild, The",
    ["Shakespeare, William"] = "Tempest, The"
};

Console.WriteLine("Author and Title List");
Console.WriteLine();
Console.WriteLine($"|{"Author",-25}|{"Title",30}|");
foreach (var title in titles)
{
    Console.WriteLine($"|{title.Key,-25}|{title.Value,30}|");
}
// Output:
// Author and Title List
//
// |Author                   |                         Title|
// |Doyle, Arthur Conan      |Hound of the Baskervilles, The|
// |London, Jack             |         Call of the Wild, The|
// |Shakespeare, William     |                 Tempest, The|
```

If the *width* value is positive, the formatted expression result is right-aligned; if negative, it's left-aligned. Remove the `-` signs before the width specifier and run the sample again to see the results.

If you need to specify both width and a format string, start with the width component:

C#

```
{<interpolationExpression>,<width>:<formatString>}
```

The following example shows how to specify width and alignment, and uses pipe characters ("|") to delimit text fields:

C#

```
const int NameAlignment = -9;
const int ValueAlignment = 7;
double a = 3;
double b = 4;
Console.WriteLine($"Three classical Pythagorean means of {a} and {b}:");
Console.WriteLine($"|{"Arithmetic",NameAlignment}|{0.5 * (a +
b),ValueAlignment:F3}|");
Console.WriteLine($"|{"Geometric",NameAlignment}|{Math.Sqrt(a *
b),ValueAlignment:F3}|");
```

```
Console.WriteLine($"|{"Harmonic",NameAlignment}|{2 / (1 / a + 1 /
b),ValueAlignment:F3}|");
// Output:
// Three classical Pythagorean means of 3 and 4:
// |Arithmetic|  3.500|
// |Geometric|  3.464|
// |Harmonic |  3.429|
```

As the example output shows, if the length of the formatted expression result exceeds specified field width, the *width* value is ignored.

For more information, see the Width component section of the Composite formatting article.

How to use escape sequences in an interpolated string

Interpolated strings support all escape sequences that can be used in ordinary string literals. For more information, see String escape sequences.

To interpret escape sequences literally, use a verbatim string literal. An interpolated verbatim string starts with both the `$` and `@` characters. You can use `$` and `@` in any order: both `$@"..."` and `@$"..."` are valid interpolated verbatim strings.

To include a brace, "{" or "}", in a result string, use two braces, "{{" or "}}". For more information, see the Escaping braces section of the Composite formatting article.

The following example shows how to include braces in a result string and construct a verbatim interpolated string:

C#

```
var xs = new int[] { 1, 2, 7, 9 };
var ys = new int[] { 7, 9, 12 };
Console.WriteLine($"Find the intersection of the {{{string.Join(", ",xs)}}} and
{{{string.Join(", ",ys)}}} sets.");
// Output:
// Find the intersection of the {1, 2, 7, 9} and {7, 9, 12} sets.

var userName = "Jane";
var stringWithEscapes = $"C:\\Users\\{userName}\\Documents";
var verbatimInterpolated = $@"C:\Users\{userName}\Documents";
Console.WriteLine(stringWithEscapes);
Console.WriteLine(verbatimInterpolated);
// Output:
// C:\Users\Jane\Documents
// C:\Users\Jane\Documents
```

You can use interpolated raw string literals.

How to use a ternary conditional operator `?:` in an interpolation expression

As the colon (":") has special meaning in an item with an interpolation expression, in order to use a conditional operator in an expression, enclose it in parentheses, as the following example shows:

C#

```
var rand = new Random();
for (int i = 0; i < 7; i++)
{
    Console.WriteLine($"Coin flip: {(rand.NextDouble() < 0.5 ? "heads" :
"tails")}");
}
```

How to create a culture-specific result string with string interpolation

By default, an interpolated string uses the current culture defined by the CultureInfo.CurrentCulture property for all formatting operations.

Beginning with .NET 6, you can use the String.Create(IFormatProvider, DefaultInterpolatedStringHandler) method to resolve an interpolated string to a culture-specific result string, as the following example shows:

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
foreach (var culture in cultures)
{
    var cultureSpecificMessage = string.Create(culture, $"{date,23}{number,20:N3}");
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
```

```
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
// nl-NL         27-08-2023 12:35:31      31.415.926,536
//               08/27/2023 12:35:31      31,415,926.536
```

In earlier versions of .NET, use implicit conversion of an interpolated string to a System.FormattableString instance and call its ToString(IFormatProvider) method to create a culture-specific result string. The following example shows how to do that:

C#

```
var cultures = new System.Globalization.CultureInfo[]
{
    System.Globalization.CultureInfo.GetCultureInfo("en-US"),
    System.Globalization.CultureInfo.GetCultureInfo("en-GB"),
    System.Globalization.CultureInfo.GetCultureInfo("nl-NL"),
    System.Globalization.CultureInfo.InvariantCulture
};
var date = DateTime.Now;
var number = 31_415_926.536;
FormattableString message = $"{date,23}{number,20:N3}";
foreach (var culture in cultures)
{
    var cultureSpecificMessage = message.ToString(culture);
    Console.WriteLine($"{culture.Name,-10}{cultureSpecificMessage}");
}
// Output is similar to:
// en-US       8/27/2023 12:35:31 PM      31,415,926.536
// en-GB         27/08/2023 12:35:31      31,415,926.536
// nl-NL         27-08-2023 12:35:31      31.415.926,536
//               08/27/2023 12:35:31      31,415,926.536
```

As the example shows, you can use one FormattableString instance to generate multiple result strings for various cultures.

How to create a result string using the invariant culture

Beginning with .NET 6, use the String.Create(IFormatProvider, DefaultInterpolatedStringHandler) method to resolve an interpolated string to a result string for the InvariantCulture, as the following example shows:

C#

```
string message = string.Create(CultureInfo.InvariantCulture, $"Date and time in
invariant culture: {DateTime.Now}");
Console.WriteLine(message);
```

```
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

In earlier versions of .NET, along with the FormattableString.ToString(IFormatProvider) method, you can use the static FormattableString.Invariant method, as the following example shows:

C#

```
string message = FormattableString.Invariant($"Date and time in invariant culture:
{DateTime.Now}");
Console.WriteLine(message);
// Output is similar to:
// Date and time in invariant culture: 05/17/2018 15:46:24
```

Conclusion

This tutorial describes common scenarios of string interpolation usage. For more information about string interpolation, see String interpolation. For more information about formatting types in .NET, see the Formatting types in .NET and Composite formatting articles.

See also

- String.Format
- System.FormattableString
- System.IFormattable
- Strings

Last updated on 11/18/2025

Console app

Article • 03/14/2023

This tutorial teaches you a number of features in .NET and the C# language. You'll learn:

- The basics of the .NET CLI
- The structure of a C# Console Application
- Console I/O
- The basics of File I/O APIs in .NET
- The basics of the Task-based Asynchronous Programming in .NET

You'll build an application that reads a text file, and echoes the contents of that text file to the console. The output to the console is paced to match reading it aloud. You can speed up or slow down the pace by pressing the '<' (less than) or '>' (greater than) keys. You can run this application on Windows, Linux, macOS, or in a Docker container.

There are a lot of features in this tutorial. Let's build them one by one.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Create the app

The first step is to create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console` at the command prompt. This creates the starter files for a basic "Hello World" application.

Before you start making modifications, let's run the simple Hello World application. After creating the application, type `dotnet run` at the command prompt. This command runs the NuGet package restore process, creates the application executable, and runs the executable.

The simple Hello World application code is all in *Program.cs*. Open that file with your favorite text editor. Replace the code in *Program.cs* with the following code:

```
C#
```

```
namespace TeleprompterConsole;

internal class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

At the top of the file, see a `namespace` statement. Like other Object Oriented languages you may have used, C# uses namespaces to organize types. This Hello World program is no different. You can see that the program is in the namespace with the name `TeleprompterConsole`.

Reading and Echoing the File

The first feature to add is the ability to read a text file and display all that text to the console. First, let's add a text file. Copy the sampleQuotes.txt file from the GitHub repository for this sample into your project directory. This will serve as the script for your application. For information on how to download the sample app for this tutorial, see the instructions in Samples and Tutorials.

Next, add the following method in your `Program` class (right below the `Main` method):

C#

```
static IEnumerable<string> ReadFrom(string file)
{
    string? line;
    using (var reader = File.OpenText(file))
    {
        while ((line = reader.ReadLine()) != null)
        {
            yield return line;
        }
    }
}
```

This method is a special type of C# method called an *iterator method*. Iterator methods return sequences that are evaluated lazily. That means each item in the sequence is generated as it is requested by the code consuming the sequence. Iterator methods are methods that contain one or more yield return statements. The object returned by the `ReadFrom` method contains the code to generate each item in the sequence. In this example, that involves reading the next line of text from the source file, and returning

that string. Each time the calling code requests the next item from the sequence, the code reads the next line of text from the file and returns it. When the file is completely read, the sequence indicates that there are no more items.

There are two C# syntax elements that may be new to you. The using statement in this method manages resource cleanup. The variable that is initialized in the `using` statement (`reader`, in this example) must implement the IDisposable interface. That interface defines a single method, `Dispose`, that should be called when the resource should be released. The compiler generates that call when execution reaches the closing brace of the `using` statement. The compiler-generated code ensures that the resource is released even if an exception is thrown from the code in the block defined by the using statement.

The `reader` variable is defined using the `var` keyword. var defines an *implicitly typed local variable*. That means the type of the variable is determined by the compile-time type of the object assigned to the variable. Here, that is the return value from the OpenText(String) method, which is a StreamReader object.

Now, let's fill in the code to read the file in the `Main` method:

C#

```
var lines = ReadFrom("sampleQuotes.txt");
foreach (var line in lines)
{
    Console.WriteLine(line);
}
```

Run the program (using `dotnet run`) and you can see every line printed out to the console.

Adding Delays and Formatting output

What you have is being displayed far too fast to read aloud. Now you need to add the delays in the output. As you start, you'll be building some of the core code that enables asynchronous processing. However, these first steps will follow a few anti-patterns. The anti-patterns are pointed out in comments as you add the code, and the code will be updated in later steps.

There are two steps to this section. First, you'll update the iterator method to return single words instead of entire lines. That's done with these modifications. Replace the `yield return line;` statement with the following code:

C#

```
var words = line.Split(' ');
foreach (var word in words)
{
    yield return word + " ";
}
yield return Environment.NewLine;
```

Next, you need to modify how you consume the lines of the file, and add a delay after writing each word. Replace the `Console.WriteLine(line)` statement in the `Main` method with the following block:

C#

```
Console.Write(line);
if (!string.IsNullOrWhiteSpace(line))
{
    var pause = Task.Delay(200);
    // Synchronously waiting on a task is an
    // anti-pattern. This will get fixed in later
    // steps.
    pause.Wait();
}
```

Run the sample, and check the output. Now, each single word is printed, followed by a 200 ms delay. However, the displayed output shows some issues because the source text file has several lines that have more than 80 characters without a line break. That can be hard to read while it's scrolling by. That's easy to fix. You'll just keep track of the length of each line, and generate a new line whenever the line length reaches a certain threshold. Declare a local variable after the declaration of `words` in the `ReadFrom` method that holds the line length:

C#

```
var lineLength = 0;
```

Then, add the following code after the `yield return word + " ";` statement (before the closing brace):

C#

```
lineLength += word.Length + 1;
if (lineLength > 70)
{
    yield return Environment.NewLine;
```

```
        lineLength = 0;
    }
```

Run the sample, and you'll be able to read aloud at its pre-configured pace.

Async Tasks

In this final step, you'll add the code to write the output asynchronously in one task, while also running another task to read input from the user if they want to speed up or slow down the text display, or stop the text display altogether. This has a few steps in it and by the end, you'll have all the updates that you need. The first step is to create an asynchronous Task returning method that represents the code you've created so far to read and display the file.

Add this method to your `Program` class (it's taken from the body of your `Main` method):

C#

```
private static async Task ShowTeleprompter()
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(200);
        }
    }
}
```

You'll notice two changes. First, in the body of the method, instead of calling Wait() to synchronously wait for a task to finish, this version uses the `await` keyword. In order to do that, you need to add the `async` modifier to the method signature. This method returns a `Task`. Notice that there are no return statements that return a `Task` object. Instead, that `Task` object is created by code the compiler generates when you use the `await` operator. You can imagine that this method returns when it reaches an `await`. The returned `Task` indicates that the work has not completed. The method resumes when the awaited task completes. When it has executed to completion, the returned `Task` indicates that it is complete. Calling code can monitor that returned `Task` to determine when it has completed.

Add an `await` keyword before the call to `ShowTeleprompter`:

C#

```
await ShowTeleprompter();
```

This requires you to change the `Main` method signature to:

C#

```
static async Task Main(string[] args)
```

Learn more about the async Main method in our fundamentals section.

Next, you need to write the second asynchronous method to read from the Console and watch for the '<' (less than), '>' (greater than) and 'X' or 'x' keys. Here's the method you add for that task:

C#

```
private static async Task GetInput()
{
    var delay = 200;
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
            {
                delay -= 10;
            }
            else if (key.KeyChar == '<')
            {
                delay += 10;
            }
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
            {
                break;
            }
        } while (true);
    };
    await Task.Run(work);
}
```

This creates a lambda expression to represent an Action delegate that reads a key from the Console and modifies a local variable representing the delay when the user presses the '<' (less than) or '>' (greater than) keys. The delegate method finishes when user presses the 'X' or 'x' keys, which allow the user to stop the text display at any time. This method uses ReadKey() to block and wait for the user to press a key.

To finish this feature, you need to create a new `async Task` returning method that starts both of these tasks (`GetInput` and `ShowTeleprompter`), and also manages the shared data between these two tasks.

It's time to create a class that can handle the shared data between these two tasks. This class contains two public properties: the delay, and a flag `Done` to indicate that the file has been completely read:

C#

```
namespace TeleprompterConsole;

internal class TelePrompterConfig
{
    public int DelayInMilliseconds { get; private set; } = 200;
    public void UpdateDelay(int increment) // negative to speed up
    {
        var newDelay = Min(DelayInMilliseconds + increment, 1000);
        newDelay = Max(newDelay, 20);
        DelayInMilliseconds = newDelay;
    }
    public bool Done { get; private set; }
    public void SetDone()
    {
        Done = true;
    }
}
```

Put that class in a new file, and include that class in the `TeleprompterConsole` namespace as shown. You'll also need to add a `using static` statement at the top of the file so that you can reference the `Min` and `Max` methods without the enclosing class or namespace names. A using static statement imports the methods from one class. This is in contrast with the `using` statement without `static`, which imports all classes from a namespace.

C#

```
using static System.Math;
```

Next, you need to update the `ShowTeleprompter` and `GetInput` methods to use the new `config` object. Write one final `Task` returning `async` method to start both tasks and exit when the first task finishes:

C#

```
private static async Task RunTeleprompter()
{
```

```
    var config = new TelePrompterConfig();
    var displayTask = ShowTeleprompter(config);

    var speedTask = GetInput(config);
    await Task.WhenAny(displayTask, speedTask);
}
```

The one new method here is the WhenAny(Task[]) call. That creates a `Task` that finishes as soon as any of the tasks in its argument list completes.

Next, you need to update both the `ShowTeleprompter` and `GetInput` methods to use the `config` object for the delay:

C#

```
private static async Task ShowTeleprompter(TelePrompterConfig config)
{
    var words = ReadFrom("sampleQuotes.txt");
    foreach (var word in words)
    {
        Console.Write(word);
        if (!string.IsNullOrWhiteSpace(word))
        {
            await Task.Delay(config.DelayInMilliseconds);
        }
    }
    config.SetDone();
}

private static async Task GetInput(TelePrompterConfig config)
{
    Action work = () =>
    {
        do {
            var key = Console.ReadKey(true);
            if (key.KeyChar == '>')
                config.UpdateDelay(-10);
            else if (key.KeyChar == '<')
                config.UpdateDelay(10);
            else if (key.KeyChar == 'X' || key.KeyChar == 'x')
                config.SetDone();
        } while (!config.Done);
    };
    await Task.Run(work);
}
```

This new version of `ShowTeleprompter` calls a new method in the `TelepromptConfig` class. Now, you need to update `Main` to call `RunTeleprompter` instead of `ShowTeleprompter`:

C#

```
await RunTeleprompter();
```

Conclusion

This tutorial showed you a number of the features around the C# language and the .NET Core libraries related to working in Console applications. You can build on this knowledge to explore more about the language, and the classes introduced here. You've seen the basics of File and Console I/O, blocking and non-blocking use of the Task-based asynchronous programming, a tour of the C# language and how C# programs are organized, and the .NET CLI.

For more information about File I/O, see File and Stream I/O. For more information about asynchronous programming model used in this tutorial, see Task-based Asynchronous Programming and Asynchronous programming.

Tutorial: Make HTTP requests in a .NET console app using C#

This tutorial builds an app that issues HTTP requests to a REST service on GitHub. The app reads information in JSON format and converts the JSON into C# objects. Converting from JSON to C# objects is known as *deserialization*.

The tutorial shows how to:

- ✓ Send HTTP requests.
- ✓ Deserialize JSON responses.
- ✓ Configure deserialization with attributes.

If you prefer to follow along with the final sample for this tutorial, you can download it. For download instructions, see Samples and Tutorials.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Create the client app

1. Open a command prompt and create a new directory for your app. Make that the current directory.

2. Enter the following command in a console window:

   .NET CLI

   ```
   dotnet new console --name WebAPIClient
   ```

   This command creates the starter files for a basic "Hello World" app. The project name is "WebAPIClient".

3. Navigate into the "WebAPIClient" directory, and run the app.

   .NET CLI

   ```
   cd WebAPIClient
   ```

.NET CLI

```
dotnet run
```

dotnet run automatically runs dotnet restore to restore any dependencies that the app needs. It also runs dotnet build if needed. You should see the app output `"Hello, World!"`. In your terminal, press `Ctrl` + `C` to stop the app.

Make HTTP requests

This app calls the GitHub API to get information about the projects under the .NET Foundation umbrella. The endpoint is https://api.github.com/orgs/dotnet/repos. To retrieve information, it makes an HTTP GET request. Browsers also make HTTP GET requests, so you can paste that URL into your browser address bar to see what information you'll be receiving and processing.

Use the HttpClient class to make HTTP requests. HttpClient supports only async methods for its long-running APIs. So the following steps create an async method and call it from the Main method.

1. Open the `Program.cs` file in your project directory and replace its contents with the following:

   C#

   ```
   await ProcessRepositoriesAsync();

   static async Task ProcessRepositoriesAsync(HttpClient client)
   {
   }
   ```

   This code:

   - Replaces the `Console.WriteLine` statement with a call to `ProcessRepositoriesAsync` that uses the `await` keyword.
   - Defines an empty `ProcessRepositoriesAsync` method.

2. In the `Program` class, use an HttpClient to handle requests and responses, by replacing the content with the following C#.

   C#

   ```
   using System.Net.Http.Headers;

   using HttpClient client = new();
   ```

```
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository
Reporter");

await ProcessRepositoriesAsync(client);

static async Task ProcessRepositoriesAsync(HttpClient client)
{
}
```

This code:

- Sets up HTTP headers for all requests:
  - An Accept header to accept JSON responses
  - A User-Agent header. These headers are checked by the GitHub server code and are necessary to retrieve information from GitHub.

3. In the `ProcessRepositoriesAsync` method, call the GitHub endpoint that returns a list of all repositories under the .NET foundation organization:

C#

```
static async Task ProcessRepositoriesAsync(HttpClient client)
{
    var json = await client.GetStringAsync(
        "https://api.github.com/orgs/dotnet/repos");

    Console.Write(json);
}
```

This code:

- Awaits the task returned from calling HttpClient.GetStringAsync(String) method. This method sends an HTTP GET request to the specified URI. The body of the response is returned as a String, which is available when the task completes.
- The response string `json` is printed to the console.

4. Build the app and run it.

.NET CLI

```
dotnet run
```

There is no build warning because the `ProcessRepositoriesAsync` now contains an `await` operator. The output is a long display of JSON text.

Deserialize the JSON Result

The following steps simplify the approach to fetching the data and processing it. You will use the GetFromJsonAsync extension method that's part of the System.Net.Http.Json NuGet package to fetch and deserialize the JSON results into objects.

1. Create a file named *Repository.cs* and add the following code:

   C#

   ```
   public record class Repository(string Name);
   ```

   The preceding code defines a class to represent the JSON object returned from the GitHub API. You'll use this class to display a list of repository names.

   The JSON for a repository object contains dozens of properties, but only the `Name` property will be deserialized. The serializer automatically ignores JSON properties for which there is no match in the target class. This feature makes it easier to create types that work with only a subset of fields in a large JSON packet.

   Although the `GetFromJsonAsync` method you will use in the next point has a benefit of being case-insensitive when it comes to property names, the C# convention is to capitalize the first letter of property names.

2. Use the HttpClientJsonExtensions.GetFromJsonAsync method to fetch and convert JSON into C# objects. Replace the call to GetStringAsync(String) in the `ProcessRepositoriesAsync` method with the following lines:

   C#

   ```
   var repositories = await client.GetFromJsonAsync<List<Repository>>
   ("https://api.github.com/orgs/dotnet/repos");
   ```

   The updated code replaces GetStringAsync(String) with HttpClientJsonExtensions.GetFromJsonAsync.

   The first argument to `GetFromJsonAsync` method is an `await` expression. `await` expressions can appear almost anywhere in your code, even though up to now, you've only seen them as part of an assignment statement. The next parameter, `requestUri` is optional and doesn't have to be provided if was already specified when creating the `client` object. You didn't provide the `client` object with the URI to send request to, so you specified the URI now. The last optional parameter, the `CancellationToken` is omitted in the code snippet.

The `GetFromJsonAsync` method is *generic*, which means you supply type arguments for what kind of objects should be created from the fetched JSON text. In this example, you're deserializing to a `List<Repository>`, which is another generic object, a System.Collections.Generic.List<T>. The `List<T>` class stores a collection of objects. The type argument declares the type of objects stored in the `List<T>`. The type argument is your `Repository` record, because the JSON text represents a collection of repository objects.

3. Add code to display the name of each repository. Replace the lines that read:

   C#

   ```
   Console.Write(json);
   ```

   with the following code:

   C#

   ```
   foreach (var repo in repositories ?? Enumerable.Empty<Repository>())
       Console.WriteLine(repo.Name);
   ```

4. The following `using` directives should be present at the top of the file:

   C#

   ```
   using System.Net.Http.Headers;
   using System.Net.Http.Json;
   ```

5. Run the app.

   .NET CLI

   ```
   dotnet run
   ```

   The output is a list of the names of the repositories that are part of the .NET Foundation.

Refactor the code

The `ProcessRepositoriesAsync` method can do the async work and return a collection of the repositories. Change that method to return `Task<List<Repository>>`, and move the code that writes to the console near its caller.

1. Change the signature of `ProcessRepositoriesAsync` to return a task whose result is a list of `Repository` objects:

   ```
   C#
   static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient client)
   ```

2. Return the repositories after processing the JSON response:

   ```
   C#
   var repositories = await client.GetFromJsonAsync<List<Repository>>
   ("https://api.github.com/orgs/dotnet/repos");
   return repositories ?? new();
   ```

   The compiler generates the `Task<T>` object for the return value because you've marked this method as `async`.

3. Modify the *Program.cs* file, replacing the call to `ProcessRepositoriesAsync` with the following to capture the results and write each repository name to the console.

   ```
   C#
   var repositories = await ProcessRepositoriesAsync(client);

   foreach (var repo in repositories)
       Console.WriteLine(repo.Name);
   ```

4. Run the app.

   The output is the same.

Deserialize more properties

In the following steps, we extend the code to process more properties from the JSON payload returned by the GitHub API. You probably won't need to process every property, but adding a few demonstrates additional C# features.

1. Replace the contents of the `Repository` class with the following `record` definition. Make sure to import the `System.Text.Json.Serialization` namespace and apply the `[JsonPropertyName]` attribute to map JSON fields to C# properties explicitly.

   ```
   C#
   ```

```
using System.Text.Json.Serialization;

public record class Repository(
  string Name,
  string Description,
  [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
  Uri Homepage,
  int Watchers,
  [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc
 );
```

The Uri and `int` types have built-in functionality to convert to and from string representation. No extra code is needed to deserialize from JSON string format to those target types. If the JSON packet contains data that doesn't convert to a target type, the serialization action throws an exception.

JSON often uses `lowercase` or `snake_case` for property names. Fields like `html_url` and `pushed_at` do not follow C# PascalCase naming conventions. Using `[JsonPropertyName]` ensures that these JSON keys are correctly bound to their corresponding C# properties, even when their names differ in case or contain underscores. This approach guarantees predictable and stable deserialization while allowing PascalCase property names in C#. Additionally, the `GetFromJsonAsync` method is `case-insensitive` when matching property names, so no further conversion is necessary.

2. Update the `foreach` loop in the *Program.cs* file to display the property values:

C#

```
foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine();
}
```

3. Run the app.

   The list now includes the additional properties.

Add a date property

The date of the last push operation is formatted in this fashion in the JSON response:

JSON

```
2016-02-08T21:27:00Z
```

This format is for Coordinated Universal Time (UTC), so the result of deserialization is a DateTime value whose Kind property is Utc.

To get a date and time represented in your time zone, you have to write a custom conversion method.

1. In *Repository.cs*, add a property for the UTC representation of the date and time and a readonly `LastPush` property that returns the date converted to local time, the file should look like the following:

   C#

   ```
   using System.Text.Json.Serialization;

   public record class Repository(
       string Name,
       string Description,
       [property: JsonPropertyName("html_url")] Uri GitHubHomeUrl,
       Uri Homepage,
       int Watchers,
       [property: JsonPropertyName("pushed_at")] DateTime LastPushUtc
       )
   {
       public DateTime LastPush => LastPushUtc.ToLocalTime();
   }
   ```

   The `LastPush` property is defined using an *expression-bodied member* for the `get` accessor. There's no `set` accessor. Omitting the `set` accessor is one way to define a *read-only* property in C#. (Yes, you can create *write-only* properties in C#, but their value is limited.)

2. Add another output statement in *Program.cs*: again:

   C#

   ```
   Console.WriteLine($"Last push: {repo.LastPush}");
   ```

3. The complete app should resemble the following *Program.cs* file:

   C#

   ```
   using System.Net.Http.Headers;
   using System.Net.Http.Json;
   ```

```
using HttpClient client = new();
client.DefaultRequestHeaders.Accept.Clear();
client.DefaultRequestHeaders.Accept.Add(
    new MediaTypeWithQualityHeaderValue("application/vnd.github.v3+json"));
client.DefaultRequestHeaders.Add("User-Agent", ".NET Foundation Repository
Reporter");

var repositories = await ProcessRepositoriesAsync(client);

foreach (var repo in repositories)
{
    Console.WriteLine($"Name: {repo.Name}");
    Console.WriteLine($"Homepage: {repo.Homepage}");
    Console.WriteLine($"GitHub: {repo.GitHubHomeUrl}");
    Console.WriteLine($"Description: {repo.Description}");
    Console.WriteLine($"Watchers: {repo.Watchers:#,0}");
    Console.WriteLine($"{repo.LastPush}");
    Console.WriteLine();
}

static async Task<List<Repository>> ProcessRepositoriesAsync(HttpClient client)
{
    var repositories = await client.GetFromJsonAsync<List<Repository>>
("https://api.github.com/orgs/dotnet/repos");
    return repositories ?? new List<Repository>();
}
```

4. Run the app.

   The output includes the date and time of the last push to each repository.

Next steps

In this tutorial, you created an app that makes web requests and parses the results. Your version of the app should now match the finished sample.

Learn more about how to configure JSON serialization in How to serialize and deserialize (marshal and unmarshal) JSON in .NET.

Last updated on 10/28/2025

Work with Language-Integrated Query (LINQ)

Introduction

This tutorial teaches you features in .NET and the C# language. You learn how to:

- Generate sequences with LINQ.
- Write methods that you can easily use in LINQ queries.
- Distinguish between eager and lazy evaluation.

You learn these techniques by building an application that demonstrates one of the basic skills of any magician: the faro shuffle. A faro shuffle is a technique where you split a card deck exactly in half, and then the shuffle interleaves each card from each half to rebuild the original deck.

Magicians use this technique because every card is in a known location after each shuffle, and the order is a repeating pattern.

This tutorial offers a lighthearted look at manipulating sequences of data. The application constructs a card deck, performs a sequence of shuffles, and writes the sequence out each time. It also compares the updated order to the original order.

This tutorial has multiple steps. After each step, you can run the application and see the progress. You can also see the completed sample in the dotnet/samples GitHub repository. For download instructions, see Samples and Tutorials.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Create the application

Create a new application. Open a command prompt and create a new directory for your application. Make that the current directory. Type the command `dotnet new console -o LinqFaroShuffle` at the command prompt. This command creates the starter files for a basic "Hello World" application.

If you've never used C# before, this tutorial explains the structure of a C# program. You can read that and then return here to learn more about LINQ.

Create the data set

> **Tip**
>
> For this tutorial, you can organize your code in a namespace called `LinqFaroShuffle` to match the sample code, or you can use the default global namespace. If you choose to use a namespace, make sure all your classes and methods are consistently within the same namespace, or add appropriate `using` statements as needed.

Consider what constitutes a deck of cards. A deck of playing cards has four suits, and each suit has 13 values. Normally, you might consider creating a `Card` class right away and populating a collection of `Card` objects by hand. With LINQ, you can be more concise than the usual way of creating a deck of cards. Instead of creating a `Card` class, create two sequences to represent suits and ranks. Create a pair of *iterator methods* that generate the ranks and suits as IEnumerable<T>s of strings:

C#

```
static IEnumerable<string> Suits()
{
    yield return "clubs";
    yield return "diamonds";
    yield return "hearts";
    yield return "spades";
}

static IEnumerable<string> Ranks()
{
    yield return "two";
    yield return "three";
    yield return "four";
    yield return "five";
    yield return "six";
    yield return "seven";
    yield return "eight";
    yield return "nine";
    yield return "ten";
    yield return "jack";
    yield return "queen";
    yield return "king";
    yield return "ace";
}
```

Place these methods under the `Console.WriteLine` statement in your `Program.cs` file. These two methods both use the `yield return` syntax to produce a sequence as they run. The compiler builds an object that implements IEnumerable<T> and generates the sequence of strings as they're requested.

Now, use these iterator methods to create the deck of cards. Place the LINQ query at the top of the `Program.cs` file. Here's what it looks like:

C#

```
var startingDeck = from s in Suits()
                   from r in Ranks()
                   select (Suit: s, Rank: r);

// Display each card that's generated and placed in startingDeck
foreach (var card in startingDeck)
{
    Console.WriteLine(card);
}
```

The multiple `from` clauses produce a SelectMany, which creates a single sequence from combining each element in the first sequence with each element in the second sequence. The order is important for this example. The first element in the first source sequence (Suits) is combined with every element in the second sequence (Ranks). This process produces all 13 cards of first suit. That process is repeated with each element in the first sequence (Suits). The end result is a deck of cards ordered by suits, followed by values.

Keep in mind that whether you write your LINQ in the query syntax used in the preceding sample or use method syntax instead, it's always possible to go from one form of syntax to the other. The preceding query written in query syntax can be written in method syntax as:

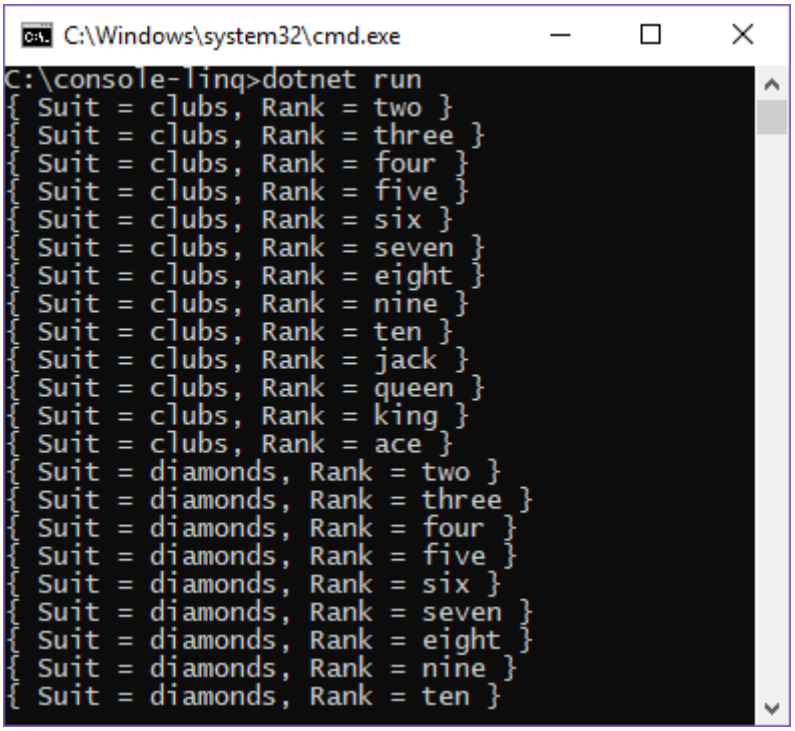
C#

```
var startingDeck = Suits().SelectMany(suit => Ranks().Select(rank => (Suit: suit,
Rank: rank )));
```

The compiler translates LINQ statements written with query syntax into the equivalent method call syntax. Therefore, regardless of your syntax choice, the two versions of the query produce the same result. Choose the syntax that works best for your situation. For instance, if you're working in a team where some members have difficulty with method syntax, try to prefer using query syntax.

Run the sample you built at this point. It displays all 52 cards in the deck. You might find it helpful to run this sample under a debugger to observe how the `Suits()` and `Ranks()`

methods execute. You can clearly see that each string in each sequence is generated only as needed.



Manipulate the order

Next, focus on how you shuffle the cards in the deck. The first step in any good shuffle is to split the deck in two. The Take and Skip methods that are part of the LINQ APIs provide that feature. Place them following the `foreach` loop:

C#

```
var top = startingDeck.Take(26);
var bottom = startingDeck.Skip(26);
```

However, there's no shuffle method to take advantage of in the standard library, so you need to write your own. The shuffle method you create illustrates several techniques that you use with LINQ-based programs, so each part of this process is explained in steps.

To add functionality to how you interact with the IEnumerable<T> results of LINQ queries, you write some special kinds of methods called extension methods. An extension method is a special purpose *static method* that adds new functionality to an already-existing type without having to modify the original type you want to add functionality to.

Give your extension methods a new home by adding a new *static* class file to your program called `Extensions.cs`, and then start building out the first extension method:

C#

```
public static class CardExtensions
{
    extension<T>(IEnumerable<T> sequence)
    {
        public IEnumerable<T> InterleaveSequenceWith(IEnumerable<T> second)
        {
            // Your implementation goes here
            return default;
        }
    }
}
```

> **Note**
>
> If you're using an editor other than Visual Studio (such as Visual Studio Code), you might need to add `using LinqFaroShuffle;` to the top of your *Program.cs* file for the extension methods to be accessible. Visual Studio automatically adds this using statement, but other editors might not.

The `extension` container specifies the type being extended. The `extension` node declares the type and name of the *receiver parameter* for all members inside the `extension` container. In this example, you're extending `IEnumerable<T>`, and the parameter is named `sequence`.

Extension member declarations appear as though they're members of the receiver type:

C#

```
public IEnumerable<T> InterleaveSequenceWith(IEnumerable<T> second)
```

You call the method as though it were a member method of the extended type. This method declaration also follows a standard idiom where the input and output types are `IEnumerable<T>`. That practice enables LINQ methods to be chained together to perform more complex queries.

Since you split the deck into halves, you need to join those halves together. In code, this means you enumerate both of the sequences you acquired through Take and Skip at once, *interleaving* the elements, and creating one sequence: your now-shuffled deck of cards. Writing a LINQ method that works with two sequences requires that you understand how IEnumerable<T> works.

The IEnumerable<T> interface has one method: GetEnumerator. The object returned by GetEnumerator has a method to move to the next element and a property that retrieves the current element in the sequence. You use those two members to enumerate the collection and return the elements. This Interleave method is an iterator method, so instead of building a

collection and returning the collection, you use the `yield return` syntax shown in the preceding code.

Here's the implementation of that method:

C#

```
public IEnumerable<T> InterleaveSequenceWith(IEnumerable<T> second)
{
    var firstIter = sequence.GetEnumerator();
    var secondIter = second.GetEnumerator();

    while (firstIter.MoveNext() && secondIter.MoveNext())
    {
        yield return firstIter.Current;
        yield return secondIter.Current;
    }
}
```

Now that you wrote this method, go back to the `Main` method and shuffle the deck once:

C#

```
var shuffledDeck = top.InterleaveSequenceWith(bottom);

foreach (var c in shuffledDeck)
{
    Console.WriteLine(c);
}
```

Comparisons

Determine how many shuffles it takes to set the deck back to its original order. To find out, write a method that determines if two sequences are equal. After you have that method, place the code that shuffles the deck in a loop, and check to see when the deck is back in order.

Writing a method to determine if the two sequences are equal should be straightforward. It's a similar structure to the method you wrote to shuffle the deck. However, this time, instead of using `yield return` for each element, you compare the matching elements of each sequence. When the entire sequence is enumerated, if every element matches, the sequences are the same:

C#

```
public bool SequenceEquals(IEnumerable<T> second)
{
    var firstIter = sequence.GetEnumerator();
```

```
    var secondIter = second.GetEnumerator();

    while ((firstIter?.MoveNext() == true) && secondIter.MoveNext())
    {
        if ((firstIter.Current is not null) &&
!firstIter.Current.Equals(secondIter.Current))
        {
            return false;
        }
    }

    return true;
}
```

This method shows a second LINQ idiom: terminal methods. They take a sequence as input (or in this case, two sequences) and return a single scalar value. When you use terminal methods, they're always the final method in a chain of methods for a LINQ query.

You can see this in action when you use it to determine when the deck is back in its original order. Put the shuffle code inside a loop, and stop when the sequence is back in its original order by applying the `SequenceEquals()` method. You can see it would always be the final method in any query because it returns a single value instead of a sequence:

C#

```
var startingDeck = from s in Suits()
                   from r in Ranks()
                   select (Suit: s, Rank: r);

// Display each card generated and placed in startingDeck in the console
foreach (var card in startingDeck)
{
    Console.WriteLine(card);
}

var top = startingDeck.Take(26);
var bottom = startingDeck.Skip(26);

var shuffledDeck = top.InterleaveSequenceWith(bottom);

var times = 0;
// Re-use the shuffle variable from earlier, or you can make a new one
shuffledDeck = startingDeck;
do
{
    shuffledDeck =
shuffledDeck.Take(26).InterleaveSequenceWith(shuffledDeck.Skip(26));

    foreach (var card in shuffledDeck)
    {
        Console.WriteLine(card);
    }
```

```
    Console.WriteLine();
    times++;

} while (!startingDeck.SequenceEquals(shuffledDeck));

Console.WriteLine(times);
```

Run the code you built so far and notice how the deck rearranges on each shuffle. After 8 shuffles (iterations of the do-while loop), the deck returns to the original configuration it was in when you first created it from the starting LINQ query.

Optimizations

The sample you built so far executes an *out shuffle*, where the top and bottom cards stay the same on each run. Let's make one change: use an *in shuffle* instead, where all 52 cards change position. For an in shuffle, you interleave the deck so that the first card in the bottom half becomes the first card in the deck. That means the last card in the top half becomes the bottom card. This change requires one line of code. Update the current shuffle query by switching the positions of Take and Skip. This change switches the order of the top and bottom halves of the deck:

C#

```
shuffledDeck = shuffledDeck.Skip(26).InterleaveSequenceWith(shuffledDeck.Take(26));
```

Run the program again, and you see that it takes 52 iterations for the deck to reorder itself. You also notice some serious performance degradation as the program continues to run.

There are several reasons for this performance drop. You can tackle one of the major causes: inefficient use of *lazy evaluation*.

Lazy evaluation states that the evaluation of a statement isn't performed until its value is needed. LINQ queries are statements that are evaluated lazily. The sequences are generated only as the elements are requested. Usually, that's a major benefit of LINQ. However, in a program like this one, lazy evaluation causes exponential growth in execution time.

Remember that you generated the original deck using a LINQ query. Each shuffle is generated by performing three LINQ queries on the previous deck. All these queries are performed lazily. That also means they're performed again each time the sequence is requested. By the time you get to the 52nd iteration, you're regenerating the original deck many times. Write a log to demonstrate this behavior. Once you gather data, you can improve performance.

In your `Extensions.cs` file, type in or copy the method in the following code sample. This extension method creates a new file called `debug.log` within your project directory and records

what query is currently being executed to the log file. Append this extension method to any query to mark that the query executed.

C#

```
public IEnumerable<T> LogQuery(string tag)
{
    // File.AppendText creates a new file if the file doesn't exist.
    using (var writer = File.AppendText("debug.log"))
    {
        writer.WriteLine($"Executing Query {tag}");
    }

    return sequence;
}
```

Next, instrument the definition of each query with a log message:

C#

```
var startingDeck = (from s in Suits().LogQuery("Suit Generation")
                    from r in Ranks().LogQuery("Rank Generation")
                    select (Suit: s, Rank: r)).LogQuery("Starting Deck");

foreach (var c in startingDeck)
{
    Console.WriteLine(c);
}

Console.WriteLine();
var times = 0;
var shuffle = startingDeck;

do
{
    // Out shuffle
    /*
    shuffle = shuffle.Take(26)
        .LogQuery("Top Half")
        .InterleaveSequenceWith(shuffle.Skip(26)
        .LogQuery("Bottom Half"))
        .LogQuery("Shuffle");
    */

    // In shuffle
    shuffle = shuffle.Skip(26).LogQuery("Bottom Half")
            .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
            .LogQuery("Shuffle");

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
```

```
    }

    times++;
    Console.WriteLine(times);
} while (!startingDeck.SequenceEquals(shuffle));

Console.WriteLine(times);
```

Notice that you don't log every time you access a query. You log only when you create the original query. The program still takes a long time to run, but now you can see why. If you run out of patience running the in shuffle with logging turned on, switch back to the out shuffle. You still see the lazy evaluation effects. In one run, it executes 2,592 queries, including the value and suit generation.

You can improve the performance of the code to reduce the number of executions you make. A simple fix is to *cache* the results of the original LINQ query that constructs the deck of cards. Currently, you're executing the queries again and again every time the do-while loop goes through an iteration, reconstructing the deck of cards and reshuffling it every time. To cache the deck of cards, apply the LINQ methods ToArray and ToList. When you append them to the queries, they perform the same actions you told them to, but now they store the results in an array or a list, depending on which method you choose to call. Append the LINQ method ToArray to both queries and run the program again:

C#

```
var startingDeck = (from s in suits().LogQuery("Suit Generation")
                    from r in ranks().LogQuery("Value Generation")
                    select new { Suit = s, Rank = r })
                    .LogQuery("Starting Deck")
                    .ToArray();

foreach (var c in startingDeck)
{
    Console.WriteLine(c);
}

Console.WriteLine();

var times = 0;
var shuffle = startingDeck;

do
{
    /*
    shuffle = shuffle.Take(26)
        .LogQuery("Top Half")
        .InterleaveSequenceWith(shuffle.Skip(26).LogQuery("Bottom Half"))
        .LogQuery("Shuffle")
        .ToArray();
    */
```

```
    shuffle = shuffle.Skip(26)
        .LogQuery("Bottom Half")
        .InterleaveSequenceWith(shuffle.Take(26).LogQuery("Top Half"))
        .LogQuery("Shuffle")
        .ToArray();

    foreach (var c in shuffle)
    {
        Console.WriteLine(c);
    }

    times++;
    Console.WriteLine(times);
} while (!startingDeck.SequenceEquals(shuffle));

Console.WriteLine(times);
```

Now the out shuffle is down to 30 queries. Run again with the in shuffle and you see similar improvements: it now executes 162 queries.

This example is **designed** to highlight the use cases where lazy evaluation can cause performance difficulties. While it's important to see where lazy evaluation can impact code performance, it's equally important to understand that not all queries should run eagerly. The performance hit you incur without using ToArray is because each new arrangement of the deck of cards is built from the previous arrangement. Using lazy evaluation means each new deck configuration is built from the original deck, even executing the code that built the `startingDeck`. That causes a large amount of extra work.

In practice, some algorithms run well using eager evaluation, and others run well using lazy evaluation. For daily usage, lazy evaluation is usually a better choice when the data source is a separate process, like a database engine. For databases, lazy evaluation allows more complex queries to execute only one round trip to the database process and back to the rest of your code. LINQ is flexible whether you choose to use lazy or eager evaluation, so measure your processes and pick whichever evaluation gives you the best performance.

Conclusion

In this project, you covered:

- Using LINQ queries to aggregate data into a meaningful sequence.
- Writing extension methods to add custom functionality to LINQ queries.
- Locating areas in code where LINQ queries might run into performance issues like degraded speed.
- Lazy and eager evaluation in LINQ queries and the implications they might have on query performance.

Aside from LINQ, you learned about a technique magicians use for card tricks. Magicians use the faro shuffle because they can control where every card moves in the deck. Now that you know, don't spoil it for everyone else!

For more information on LINQ, see:

- Introduction to LINQ
- Basic LINQ query operations (C#)
- Data transformations with LINQ (C#)
- Query syntax and method syntax in LINQ (C#)
- C# features that support LINQ

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 12/02/2025

Language Integrated Query (LINQ)

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, and events.

When you write queries, the most visible "language-integrated" part of LINQ is the query expression. You write query expressions in a declarative *query syntax*. By using query syntax, you perform filtering, ordering, and grouping operations on data sources with a minimum of code. You use the same query expression patterns to query and transform data from any type of data source.

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

C#

```
// Specify the data source.
int[] scores = [97, 92, 81, 60];

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

// Execute the query.
foreach (var i in scoreQuery)
{
    Console.Write(i + " ");
}

// Output: 97 92 81
```

You might need to add a using directive, `using System.Linq;`, for the preceding example to compile. The most recent versions of .NET make use of implicit usings to add this directive as a global using. Older versions require you to add it in your source.

Query expression overview

- Query expressions examine and 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 use many familiar C# language constructs, which make them easy to read.
- The variables in a query expression are all strongly typed.
- A query isn't executed until you iterate over the query variable, for example in a `foreach` statement.
- At compile time, the compiler converts query expressions to standard query operator method calls according to the rules defined in the C# specification. You can express any query that uses query syntax by using method syntax. In some cases, query syntax is more readable and concise. In others, method syntax is more readable. There's no semantic or performance difference between the two different forms. For more information, see C# language specification and Standard query operators overview.
- Some query operations, such as Count or Max, have no equivalent query expression clause and must be expressed as a method call. You can combine method syntax with query syntax in various ways.
- Query expressions can be compiled to expression trees or to delegates, depending on the type that the query is applied to. The compiler compiles IEnumerable<T> queries to delegates. The compiler compiles IQueryable and IQueryable<T> queries to expression trees. For more information, see Expression trees.

How to enable LINQ querying of your data source

In-memory data

You can enable LINQ querying of in-memory data in two ways. If the data is of a type that implements IEnumerable<T>, query the data by using LINQ to Objects. If it doesn't make sense to enable enumeration by implementing the IEnumerable<T> interface, define LINQ standard query operator methods either in that type or as extension members for that type. Custom implementations of the standard query operators should use deferred execution to return the results.

Remote data

The best option for enabling LINQ querying of a remote data source is to implement the IQueryable<T> interface.

IQueryable LINQ providers

LINQ providers that implement IQueryable<T> can vary widely in their complexity.

A less complex `IQueryable` provider might access a single method from a Web service. This type of provider is specific to the data source because it expects specific information in the queries that it handles. It has a closed type system, perhaps exposing a single result type. Most of the execution of the query occurs locally, for example by using the Enumerable implementations of the standard query operators. A less complex provider might examine only one method call expression in the expression tree that represents the query and let the remaining logic of the query be handled elsewhere.

An `IQueryable` provider of medium complexity might target a data source that has a partially expressive query language. If it targets a Web service, it might access more than one method of the Web service and select which method to call based on the information that the query seeks. A provider of medium complexity has a richer type system than a simple provider, but it's still a fixed type system. For example, the provider might expose types that have one-to-many relationships that can be traversed, but it doesn't provide mapping technology for user-defined types.

A complex `IQueryable` provider, such as the Entity Framework Core provider, might translate complete LINQ queries to an expressive query language, such as SQL. A complex provider is more general because it can handle a wider variety of questions in the query. It also has an open type system and therefore must contain extensive infrastructure to map user-defined types. Developing a complex provider requires a significant amount of effort.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 12/03/2025

Introduction to LINQ Queries in C#

Article • 03/24/2025

A *query* is an expression that retrieves data from a data source. Different data sources have different native query languages, for example SQL for relational databases and XQuery for XML. Developers must learn a new query language for each type of data source or data format that they must support. LINQ simplifies this situation by offering a consistent C# language model for kinds of data sources and formats. In a LINQ query, you always work with C# objects. You use the same basic coding patterns to query and transform data in XML documents, SQL databases, .NET collections, and any other format when a LINQ provider is available.

Three Parts of a Query Operation

All LINQ query operations consist of three distinct actions:

1. Obtain the data source.
2. Create the query.
3. Execute the query.

The following example shows how the three parts of a query operation are expressed in source code. The example uses an integer array as a data source for convenience; however, the same concepts apply to other data sources also. This example is referred to throughout the rest of this article.

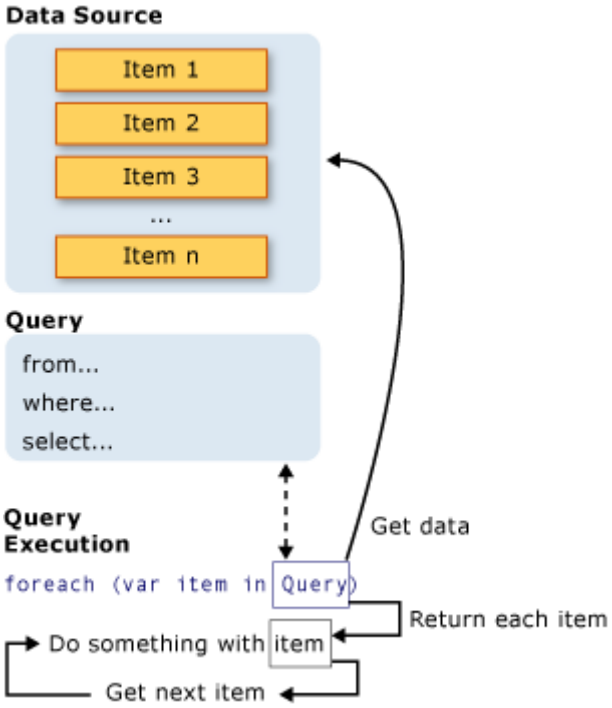
C#

```
// The Three Parts of a LINQ Query:
// 1. Data source.
int[] numbers = [ 0, 1, 2, 3, 4, 5, 6 ];

// 2. Query creation.
// numQuery is an IEnumerable<int>
var numQuery = from num in numbers
               where (num % 2) == 0
               select num;

// 3. Query execution.
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

The following illustration shows the complete query operation. In LINQ, the execution of the query is distinct from the query itself. In other words, you don't retrieve any data by creating a query variable.



The Data Source

The data source in the preceding example is an array, which supports the generic IEnumerable<T> interface. This fact means it can be queried with LINQ. A query is executed in a `foreach` statement, and `foreach` requires IEnumerable or IEnumerable<T>. Types that support IEnumerable<T> or a derived interface such as the generic IQueryable<T> are called *queryable types*.

A queryable type requires no modification or special treatment to serve as a LINQ data source. If the source data isn't already in memory as a queryable type, the LINQ provider must represent it as such. For example, LINQ to XML loads an XML document into a queryable XElement type:

C#

```
// Create a data source from an XML document.
// using System.Xml.Linq;
XElement contacts = XElement.Load(@"c:\myContactList.xml");
```

With EntityFramework, you create an object-relational mapping between C# classes and your database schema. You write your queries against the objects, and at run-time EntityFramework handles the communication with the database. In the following

example, `Customers` represents a specific table in the database, and the type of the query result, IQueryable<T>, derives from IEnumerable<T>.

C#

```
Northwnd db = new Northwnd(@"c:\northwnd.mdf");

// Query for customers in London.
IQueryable<Customer> custQuery =
    from cust in db.Customers
    where cust.City == "London"
    select cust;
```

For more information about how to create specific types of data sources, see the documentation for the various LINQ providers. However, the basic rule is simple: a LINQ data source is any object that supports the generic IEnumerable<T> interface, or an interface that inherits from it, typically IQueryable<T>.

> ⓘ **Note**
>
> Types such as **ArrayList** that support the non-generic **IEnumerable** interface can also be used as a LINQ data source. For more information, see **How to query an ArrayList with LINQ (C#)**.

The Query

The query specifies what information to retrieve from the data source or sources. Optionally, a query also specifies how that information should be sorted, grouped, and shaped before being returned. A query is stored in a query variable and initialized with a query expression. You use C# query syntax to write queries.

The query in the previous example returns all the even numbers from the integer array. The query expression contains three clauses: `from`, `where`, and `select`. (If you're familiar with SQL, you noticed that the ordering of the clauses is reversed from the order in SQL.) The `from` clause specifies the data source, the `where` clause applies the filter, and the `select` clause specifies the type of the returned elements. All the query clauses are discussed in detail in this section. For now, the important point is that in LINQ, the query variable itself takes no action and returns no data. It just stores the information that is required to produce the results when the query is executed at some later point. For more information about how queries are constructed, see Standard Query Operators Overview (C#).

> **Note**
>
> Queries can also be expressed by using method syntax. For more information, see Query Syntax and Method Syntax in LINQ.

Classification of standard query operators by manner of execution

The LINQ to Objects implementations of the standard query operator methods execute in one of two main ways: *immediate* or *deferred*. The query operators that use deferred execution can be additionally divided into two categories: *streaming* and *nonstreaming*.

Immediate

Immediate execution means that the data source is read and the operation is performed once. All the standard query operators that return a scalar result execute immediately. Examples of such queries are `Count`, `Max`, `Average`, and `First`. These methods execute without an explicit `foreach` statement because the query itself must use `foreach` in order to return a result. These queries return a single value, not an `IEnumerable` collection. You can force *any* query to execute immediately using the Enumerable.ToList or Enumerable.ToArray methods. Immediate execution provides reuse of query results, not query declaration. The results are retrieved once, then stored for future use. The following query returns a count of the even numbers in the source array:

```
C#

var evenNumQuery = from num in numbers
                   where (num % 2) == 0
                   select num;

int evenNumCount = evenNumQuery.Count();
```

To force immediate execution of any query and cache its results, you can call the ToList or ToArray methods.

```
C#

List<int> numQuery2 = (from num in numbers
                       where (num % 2) == 0
                       select num).ToList();

// or like this:
```

```
// numQuery3 is still an int[]

var numQuery3 = (from num in numbers
                 where (num % 2) == 0
                 select num).ToArray();
```

You can also force execution by putting the `foreach` loop immediately after the query expression. However, by calling `ToList` or `ToArray` you also cache all the data in a single collection object.

Deferred

Deferred execution means that the operation isn't performed at the point in the code where the query is declared. The operation is performed only when the query variable is enumerated, for example by using a `foreach` statement. The results of executing the query depend on the contents of the data source when the query is executed rather than when the query is defined. If the query variable is enumerated multiple times, the results might differ every time. Almost all the standard query operators whose return type is IEnumerable<T> or IOrderedEnumerable<TElement> execute in a deferred manner. Deferred execution provides the facility of query reuse since the query fetches the updated data from the data source each time query results are iterated. The following code shows an example of deferred execution:

C#

```
foreach (int num in numQuery)
{
    Console.Write("{0,1} ", num);
}
```

The `foreach` statement is also where the query results are retrieved. For example, in the previous query, the iteration variable `num` holds each value (one at a time) in the returned sequence.

Because the query variable itself never holds the query results, you can execute it repeatedly to retrieve updated data. For example, a separate application might update a database continually. In *your* application, you could create one query that retrieves the latest data, and you could execute it at intervals to retrieve updated results.

Query operators that use deferred execution can be additionally classified as streaming or nonstreaming.

Streaming

Streaming operators don't have to read all the source data before they yield elements. At the time of execution, a streaming operator performs its operation on each source element as it is read and yields the element if appropriate. A streaming operator continues to read source elements until a result element can be produced. This means that more than one source element might be read to produce one result element.

Nonstreaming

Nonstreaming operators must read all the source data before they can yield a result element. Operations such as sorting or grouping fall into this category. At the time of execution, nonstreaming query operators read all the source data, put it into a data structure, perform the operation, and yield the resulting elements.

Classification table

The following table classifies each standard query operator method according to its method of execution.

> **Note**
>
> If an operator is marked in two columns, two input sequences are involved in the operation, and each sequence is evaluated differently. In these cases, it is always the first sequence in the parameter list that is evaluated in a deferred, streaming manner.

Expand table

Standard query operator	Return type	Immediate execution	Deferred streaming execution	Deferred nonstreaming execution
Aggregate	`TSource`	✓		
All	Boolean	✓		
Any	Boolean	✓		
AsEnumerable	IEnumerable<T>		✓	
Average	Single numeric value	✓		
Cast	IEnumerable<T>		✓	

Standard query operator	Return type	Immediate execution	Deferred streaming execution	Deferred nonstreaming execution
Concat	IEnumerable<T>		✓	
Contains	Boolean	✓		
Count	Int32	✓		
DefaultIfEmpty	IEnumerable<T>		✓	
Distinct	IEnumerable<T>		✓	
ElementAt	`TSource`	✓		
ElementAtOrDefault	`TSource?`	✓		
Empty	IEnumerable<T>	✓		
Except	IEnumerable<T>		✓	✓
First	`TSource`	✓		
FirstOrDefault	`TSource?`	✓		
GroupBy	IEnumerable<T>			✓
GroupJoin	IEnumerable<T>		✓	✓
Intersect	IEnumerable<T>		✓	✓
Join	IEnumerable<T>		✓	✓
Last	`TSource`	✓		
LastOrDefault	`TSource?`	✓		
LongCount	Int64	✓		
Max	Single numeric value, `TSource`, or `TResult?`	✓		
Min	Single numeric value, `TSource`, or `TResult?`	✓		
OfType	IEnumerable<T>		✓	
OrderBy	IOrderedEnumerable<TElement>			✓
OrderByDescending	IOrderedEnumerable<TElement>			✓
Range	IEnumerable<T>		✓	

Standard query operator	Return type	Immediate execution	Deferred streaming execution	Deferred nonstreaming execution
Repeat	IEnumerable<T>		✓	
Reverse	IEnumerable<T>			✓
Select	IEnumerable<T>		✓	
SelectMany	IEnumerable<T>		✓	
SequenceEqual	Boolean	✓		
Single	`TSource`	✓		
SingleOrDefault	`TSource?`	✓		
Skip	IEnumerable<T>		✓	
SkipWhile	IEnumerable<T>		✓	
Sum	Single numeric value	✓		
Take	IEnumerable<T>		✓	
TakeWhile	IEnumerable<T>		✓	
ThenBy	IOrderedEnumerable<TElement>			✓
ThenByDescending	IOrderedEnumerable<TElement>			✓
ToArray	`TSource[]` array	✓		
ToDictionary	Dictionary<TKey,TValue>	✓		
ToList	IList<T>	✓		
ToLookup	ILookup<TKey,TElement>	✓		
Union	IEnumerable<T>		✓	
Where	IEnumerable<T>		✓	

LINQ to objects

"LINQ to Objects" refers to the use of LINQ queries with any IEnumerable or IEnumerable<T> collection directly. You can use LINQ to query any enumerable collections, such as List<T>, Array, or Dictionary<TKey,TValue>. The collection can be user-defined or a type returned by a .NET API. In the LINQ approach, you write

declarative code that describes what you want to retrieve. LINQ to Objects provides a great introduction to programming with LINQ.

LINQ queries offer three main advantages over traditional `foreach` loops:

- They're more concise and readable, especially when filtering multiple conditions.
- They provide powerful filtering, ordering, and grouping capabilities with a minimum of application code.
- They can be ported to other data sources with little or no modification.

The more complex the operation you want to perform on the data, the more benefit you realize using LINQ instead of traditional iteration techniques.

Store the results of a query in memory

A query is basically a set of instructions for how to retrieve and organize data. Queries are executed lazily, as each subsequent item in the result is requested. When you use `foreach` to iterate the results, items are returned as accessed. To evaluate a query and store its results without executing a `foreach` loop, just call one of the following methods on the query variable:

- ToList
- ToArray
- ToDictionary
- ToLookup

You should assign the returned collection object to a new variable when you store the query results, as shown in the following example:

C#

```
List<int> numbers = [ 1, 2, 4, 6, 8, 10, 12, 14, 16, 18, 20 ];

IEnumerable<int> queryFactorsOfFour = from num in numbers
                                      where num % 4 == 0
                                      select num;

// Store the results in a new variable
// without executing a foreach loop.
var factorsofFourList = queryFactorsOfFour.ToList();

// Read and write from the newly created list to demonstrate that it holds
data.
Console.WriteLine(factorsofFourList[2]);
```

```
factorsofFourList[2] = 0;
Console.WriteLine(factorsofFourList[2]);
```

See also

- Walkthrough: Writing Queries in C#
- foreach, in
- Query Keywords (LINQ)

Query expression basics

Article • 01/16/2025

This article introduces the basic concepts related to query expressions in C#.

What is a query and what does it do?

A *query* is a set of instructions that describes what data to retrieve from a given data source (or sources) and what shape and organization the returned data should have. A query is distinct from the results that it produces.

Generally, the source data is organized logically as a sequence of elements of the same kind. For example, an SQL database table contains a sequence of rows. In an XML file, there's a "sequence" of XML elements (although XML elements are organized hierarchically in a tree structure). An in-memory collection contains a sequence of objects.

From an application's viewpoint, the specific type and structure of the original source data isn't important. The application always sees the source data as an IEnumerable<T> or IQueryable<T> collection. For example, in LINQ to XML, the source data is made visible as an `IEnumerable`<XElement>.

Given this source sequence, a query might do one of three things:

- Retrieve a subset of the elements to produce a new sequence without modifying the individual elements. The query might then sort or group the returned sequence in various ways, as shown in the following example (assume `scores` is an `int[]`):

  C#

  ```
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  ```

- Retrieve a sequence of elements as in the previous example but transform them to a new type of object. For example, a query might retrieve only the family names from certain customer records in a data source. Or it might retrieve the complete record and then use it to construct another in-memory object type or even XML data before generating the final result sequence. The following example shows a projection from an `int` to a `string`. Note the new type of `highScoresQuery`.

```
C#

IEnumerable<string> highScoresQuery2 =
    from score in scores
    where score > 80
    orderby score descending
    select $"The score is {score}";
```

- Retrieve a singleton value about the source data, such as:
  - The number of elements that match a certain condition.
  - The element that has the greatest or least value.
  - The first element that matches a condition, or the sum of particular values in a specified set of elements. For example, the following query returns the number of scores greater than 80 from the `scores` integer array:

    ```
    C#

    var highScoreCount = (
        from score in scores
        where score > 80
        select score
    ).Count();
    ```

    In the previous example, note the use of parentheses around the query expression before the call to the Enumerable.Count method. You can also use a new variable to store the concrete result.

    ```
    C#

    IEnumerable<int> highScoresQuery3 =
        from score in scores
        where score > 80
        select score;

    var scoreCount = highScoresQuery3.Count();
    ```

In the previous example, the query is executed in the call to `Count`, because `Count` must iterate over the results in order to determine the number of elements returned by `highScoresQuery`.

What is a query expression?

A *query expression* is a query expressed in query syntax. A query expression is a first-class language construct. It's just like any other expression and can be used in any context in which a C# expression is valid. A query expression consists of a set of clauses written in a declarative syntax similar to SQL or XQuery. Each clause in turn contains one or more C# expressions, and these expressions might themselves be either a query expression or contain a query expression.

A query expression must begin with a from clause and must end with a select or group clause. Between the first `from` clause and the last `select` or `group` clause, it can contain one or more of these optional clauses: where, orderby, join, let and even another from clauses. You can also use the into keyword to enable the result of a `join` or `group` clause to serve as the source for more query clauses in the same query expression.

Query variable

In LINQ, a query variable is any variable that stores a *query* instead of the *results* of a query. More specifically, a query variable is always an enumerable type that produces a sequence of elements when iterated over in a `foreach` statement or a direct call to its IEnumerator.MoveNext() method.

> ⓘ **Note**
>
> Examples in this article use the following data source and sample data.

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City>
Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
```

```
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City",
826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro",
28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

The following code example shows a simple query expression with one data source, one filtering clause, one ordering clause, and no transformation of the source elements. The `select` clause ends the query.

C#

```
// Data source.
int[] scores = [90, 71, 82, 93, 75, 82];

// Query Expression.
IEnumerable<int> scoreQuery = //query variable
    from score in scores //required
    where score > 80 // optional
    orderby score descending // optional
    select score; //must end with select or group

// Execute the query to produce the results
foreach (var testScore in scoreQuery)
{
    Console.WriteLine(testScore);
}

// Output: 93 90 82 82
```

In the previous example, `scoreQuery` is a *query variable,* which is sometimes referred to as just a *query*. The query variable stores no actual result data, which is produced in the `foreach` loop. And when the `foreach` statement executes, the query results aren't returned through the query variable `scoreQuery`. Rather, they're returned through the iteration variable `testScore`. The `scoreQuery` variable can be iterated in a second `foreach` loop. It produces the same results as long as neither it nor the data source was modified.

A query variable might store a query that is expressed in query syntax or method syntax, or a combination of the two. In the following examples, both `queryMajorCities` and `queryMajorCities2` are query variables:

C#

```
City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000)
];

//Query syntax
IEnumerable<City> queryMajorCities =
    from city in cities
    where city.Population > 30_000_000
    select city;

// Execute the query to produce the results
foreach (City city in queryMajorCities)
{
    Console.WriteLine(city);
}

// Output:
// City { Name = Tokyo, Population = 37833000 }
// City { Name = Delhi, Population = 30290000 }

// Method-based syntax
IEnumerable<City> queryMajorCities2 = cities.Where(c => c.Population >
30_000_000);
// Execute the query to produce the results
foreach (City city in queryMajorCities2)
{
    Console.WriteLine(city);
}
// Output:
// City { Name = Tokyo, Population = 37833000 }
// City { Name = Delhi, Population = 30290000 }
```

On the other hand, the following two examples show variables that aren't query variables even though each is initialized with a query. They aren't query variables because they store results:

C#

```
var highestScore = (
    from score in scores
    select score
).Max();

// or split the expression
IEnumerable<int> scoreQuery =
    from score in scores
    select score;

var highScore = scoreQuery.Max();
// the following returns the same result
highScore = scores.Max();
```

C#

```
var largeCitiesList = (
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city
).ToList();

// or split the expression
IEnumerable<City> largeCitiesQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
var largeCitiesList2 = largeCitiesQuery.ToList();
```

Explicit and implicit typing of query variables

This documentation usually provides the explicit type of the query variable in order to show the type relationship between the query variable and the select clause. However, you can also use the var keyword to instruct the compiler to infer the type of a query variable (or any other local variable) at compile time. For example, the query example that was shown previously in this article can also be expressed by using implicit typing:

C#

```
var queryCities =
    from city in cities
    where city.Population > 100000
    select city;
```

In the preceding example, the use of var is optional. `queryCities` is an `IEnumerable<City>` whether implicitly or explicitly typed.

Starting a query expression

A query expression must begin with a `from` clause. It specifies a data source together with a range variable. The range variable represents each successive element in the source sequence as the source sequence is being traversed. The range variable is strongly typed based on the type of elements in the data source. In the following example, because `countries` is an array of `Country` objects, the range variable is also typed as `Country`. Because the range variable is strongly typed, you can use the dot operator to access any available members of the type.

C#

```
IEnumerable<Country> countryAreaQuery =
    from country in countries
    where country.Area > 20 //sq km
    select country;
```

The range variable is in scope until the query is exited either with a semicolon or with a continuation clause.

A query expression might contain multiple `from` clauses. Use more `from` clauses when each element in the source sequence is itself a collection or contains a collection. For example, assume that you have a collection of `Country` objects, each of which contains a collection of `City` objects named `Cities`. To query the `City` objects in each `Country`, use two `from` clauses as shown here:

C#

```
IEnumerable<City> cityQuery =
    from country in countries
    from city in country.Cities
    where city.Population > 10000
    select city;
```

For more information, see from clause.

Ending a query expression

A query expression must end with either a `group` clause or a `select` clause.

The group clause

Use the `group` clause to produce a sequence of groups organized by a key that you specify. The key can be any data type. For example, the following query creates a sequence of groups that contains one or more `Country` objects and whose key is a `char` type with value being the first letter of countries' names.

```
C#
```

```
var queryCountryGroups =
    from country in countries
    group country by country.Name[0];
```

For more information about grouping, see group clause.

select clause

Use the `select` clause to produce all other types of sequences. A simple `select` clause just produces a sequence of the same type of objects as the objects that are contained in the data source. In this example, the data source contains `Country` objects. The `orderby` clause just sorts the elements into a new order and the `select` clause produces a sequence of the reordered `Country` objects.

```
C#
```

```
IEnumerable<Country> sortedQuery =
    from country in countries
    orderby country.Area
    select country;
```

The `select` clause can be used to transform source data into sequences of new types. This transformation is also named a *projection*. In the following example, the `select` clause *projects* a sequence of anonymous types that contains only a subset of the fields in the original element. The new objects are initialized by using an object initializer.

```
C#
```

```
var queryNameAndPop =
    from country in countries
```

```
    select new
    {
        Name = country.Name,
        Pop = country.Population
    };
```

So in this example, the var is required because the query produces an anonymous type.

For more information about all the ways that a `select` clause can be used to transform source data, see select clause.

Continuations with *into*

You can use the `into` keyword in a `select` or `group` clause to create a temporary identifier that stores a query. Use the `into` clause when you must perform extra query operations on a query after a grouping or select operation. In the following example, `countries` are grouped according to population in ranges of 10 million. After these groups are created, more clauses filter out some groups, and then to sort the groups in ascending order. To perform those extra operations, the continuation represented by `countryGroup` is required.

C#

```
// percentileQuery is an IEnumerable<IGrouping<int, Country>>
var percentileQuery =
    from country in countries
    let percentile = (int)country.Population / 1_000
    group country by percentile into countryGroup
    where countryGroup.Key >= 20
    orderby countryGroup.Key
    select countryGroup;

// grouping is an IGrouping<int, Country>
foreach (var grouping in percentileQuery)
{
    Console.WriteLine(grouping.Key);
    foreach (var country in grouping)
    {
        Console.WriteLine(country.Name + ":" + country.Population);
    }
}
```

For more information, see into.

Filtering, ordering, and joining

Between the starting `from` clause, and the ending `select` or `group` clause, all other clauses (`where`, `join`, `orderby`, `from`, `let`) are optional. Any of the optional clauses might be used zero times or multiple times in a query body.

The where clause

Use the `where` clause to filter out elements from the source data based on one or more predicate expressions. The `where` clause in the following example has one predicate with two conditions.

C#

```
IEnumerable<City> queryCityPop =
    from city in cities
    where city.Population is < 15_000_000 and > 10_000_000
    select city;
```

For more information, see where clause.

The orderby clause

Use the `orderby` clause to sort the results in either ascending or descending order. You can also specify secondary sort orders. The following example performs a primary sort on the `country` objects by using the `Area` property. It then performs a secondary sort by using the `Population` property.

C#

```
IEnumerable<Country> querySortedCountries =
    from country in countries
    orderby country.Area, country.Population descending
    select country;
```

The `ascending` keyword is optional; it's the default sort order if no order is specified. For more information, see orderby clause.

The join clause

Use the `join` clause to associate and/or combine elements from one data source with elements from another data source based on an equality comparison between specified keys in each element. In LINQ, join operations are performed on sequences of objects whose elements are different types. After you join two sequences, you must use a

`select` or `group` statement to specify which element to store in the output sequence. You can also use an anonymous type to combine properties from each set of associated elements into a new type for the output sequence. The following example associates `prod` objects whose `Category` property matches one of the categories in the `categories` string array. Products whose `Category` doesn't match any string in `categories` are filtered out. The `select` statement projects a new type whose properties are taken from both `cat` and `prod`.

C#

```
var categoryQuery =
    from cat in categories
    join prod in products on cat equals prod.Category
    select new
    {
        Category = cat,
        Name = prod.Name
    };
```

You can also perform a group join by storing the results of the `join` operation into a temporary variable by using the into keyword. For more information, see join clause.

The let clause

Use the `let` clause to store the result of an expression, such as a method call, in a new range variable. In the following example, the range variable `firstName` stores the first element of the array of strings returned by `Split`.

C#

```
string[] names = ["Svetlana Omelchenko", "Claire O'Donnell", "Sven
Mortensen", "Cesar Garcia"];
IEnumerable<string> queryFirstNames =
    from name in names
    let firstName = name.Split(' ')[0]
    select firstName;

foreach (var s in queryFirstNames)
{
    Console.Write(s + " ");
}

//Output: Svetlana Claire Sven Cesar
```

For more information, see let clause.

Subqueries in a query expression

A query clause might itself contain a query expression, which is sometimes referred to as a *subquery*. Each subquery starts with its own `from` clause that doesn't necessarily point to the same data source in the first `from` clause. For example, the following query shows a query expression that is used in the select statement to retrieve the results of a grouping operation.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.ExamScores.Average()
        ).Max()
    };
```

For more information, see Perform a subquery on a grouping operation.

See also

- Query keywords (LINQ)
- Standard query operators overview

Write C# LINQ queries to query data

Most queries in the introductory Language Integrated Query (LINQ) documentation use the LINQ declarative query syntax. The C# compiler translates query syntax into method calls. These method calls implement the standard query operators. They have names such as `Where`, `Select`, `GroupBy`, `Join`, `Max`, and `Average`. You call them directly by using method syntax instead of query syntax.

Query syntax and method syntax are semantically identical, but query syntax is often simpler and easier to read. You must express some queries as method calls. For example, you must use a method call to express a query that retrieves the number of elements that match a specified condition. You also must use a method call for a query that retrieves the element that has the maximum value in a source sequence. The reference documentation for the standard query operators in the System.Linq namespace generally uses method syntax. Become familiar with how to use method syntax in queries and in query expressions themselves.

Standard query operator extension methods

The following example shows a simple *query expression* and the semantically equivalent query written as a *method-based query*.

C#

```
int[] numbers = [ 5, 10, 8, 3, 6, 12 ];

//Query syntax:
IEnumerable<int> numQuery1 =
    from num in numbers
    where num % 2 == 0
    orderby num
    select num;

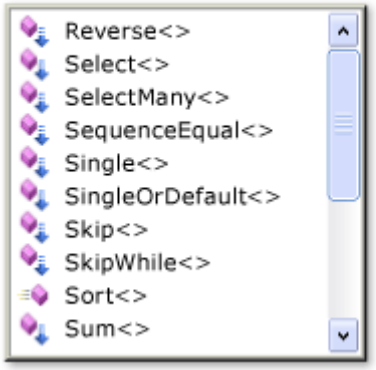
//Method syntax:
IEnumerable<int> numQuery2 = numbers
    .Where(num => num % 2 == 0)
    .OrderBy(n => n);

foreach (int i in numQuery1)
{
    Console.Write(i + " ");
}
Console.WriteLine(System.Environment.NewLine);
foreach (int i in numQuery2)
{
    Console.Write(i + " ");
}
```

The output from the two examples is identical. The type of the query variable is the same in both forms: IEnumerable<T>.

On the right side of the expression, notice that the `where` clause is now expressed as an instance method on the `numbers` object, which has a type of `IEnumerable<int>`. If you're familiar with the generic IEnumerable<T> interface, you know that it doesn't have a `Where` method. However, if you invoke the IntelliSense completion list in the Visual Studio IDE, you see not only a `Where` method, but many other methods such as `Select`, `SelectMany`, `Join`, and `Orderby`. These methods implement the standard query operators.

```
List<string> list = new List<string();
list.
```



Although it looks as if IEnumerable<T> includes more methods, it doesn't. The standard query operators are implemented as *extension methods*. Extension members "extend" an existing type; they can be called as if they were members on the type. The standard query operators extend IEnumerable<T>, and that's why you can write `numbers.Where(...)`. You bring extensions into scope with `using` directives before calling them.

For more information about extension members, see Extension members. For more information about standard query operators, see Standard Query Operators Overview (C#). Some LINQ providers, such as Entity Framework and LINQ to XML, implement their own standard query operators and extension members for other types besides IEnumerable<T>.

Lambda expressions

In the preceding example, the conditional expression (`num % 2 == 0`) is passed as an in-line argument to the Enumerable.Where method: `Where(num => num % 2 == 0).` This inline expression is a lambda expression. It's a convenient way to write code that would otherwise have to be written in more cumbersome form. The `num` on the left of the operator is the input variable, which corresponds to `num` in the query expression. The compiler can infer the type of `num` because it knows that `numbers` is a generic IEnumerable<T> type. The body of the lambda is the same as the expression in query syntax or in any other C# expression or statement. It can include method calls and other complex logic. The return value is the expression result. You can

only express certain queries in method syntax, and some of those queries require lambda expressions. Lambda expressions are a powerful and flexible tool in your LINQ toolbox.

Composability of queries

In the preceding code example, the Enumerable.OrderBy method is invoked by using the dot operator on the call to `Where`. `Where` produces a filtered sequence, and then `Orderby` sorts the sequence produced by `Where`. Because queries return an `IEnumerable`, you compose them in method syntax by chaining the method calls together. The compiler does this composition when you write queries using query syntax. Because a query variable doesn't store the results of the query, you can modify it or use it as the basis for a new query at any time, even after you execute it.

The following examples demonstrate some basic LINQ queries by using each approach listed previously.

> **Note**
>
> These queries operate on in-memory collections; however, the syntax is identical to the syntax used in LINQ to Entities and LINQ to XML.

Example - query syntax

Write most queries with *query syntax* to create *query expressions*. The following example shows three query expressions. The first query expression demonstrates how to filter or restrict results by applying conditions with a `where` clause. It returns all elements in the source sequence whose values are greater than 7 or less than 3. The second expression demonstrates how to order the returned results. The third expression demonstrates how to group results according to a key. This query returns two groups based on the first letter of the word.

C#

```
List<int> numbers = [ 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 ];

// The query variables can also be implicitly typed by using var

// Query #1.
IEnumerable<int> filteringQuery =
    from num in numbers
    where num is < 3 or > 7
    select num;

// Query #2.
```

```
IEnumerable<int> orderingQuery =
    from num in numbers
    where num is < 3 or > 7
    orderby num ascending
    select num;

// Query #3.
string[] groupingQuery = ["carrots", "cabbage", "broccoli", "beans", "barley"];
IEnumerable<IGrouping<char, string>> queryFoodGroups =
    from item in groupingQuery
    group item by item[0];
```

The type of the queries is IEnumerable<T>. All of these queries can be written using var as shown in the following example:

```
var query = from num in numbers...
```

In each previous example, the queries don't actually execute until you iterate over the query variable in a foreach statement or other statement.

Example - method syntax

You must express some query operations as a method call. The most common such methods are those methods that return singleton numeric values, such as Sum, Max, Min, Average, and so on. Call these methods last in any query because they return a single value and can't serve as the source for more query operations. The following example shows a method call in a query expression:

C#

```
List<int> numbers1 = [ 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 ];
List<int> numbers2 = [ 15, 14, 11, 13, 19, 18, 16, 17, 12, 10 ];

// Query #4.
double average = numbers1.Average();

// Query #5.
IEnumerable<int> concatenationQuery = numbers1.Concat(numbers2);
```

If the method has System.Action or System.Func<TResult> parameters, provide these arguments in the form of a lambda expression, as shown in the following example:

C#

```
// Query #6.
IEnumerable<int> largeNumbersQuery = numbers2.Where(c => c > 15);
```

In the previous queries, only Query #4 executes immediately because it returns a single value, and not a generic IEnumerable<T> collection. The method itself uses `foreach` or similar code in order to compute its value.

You can write each of the previous queries by using implicit typing with var, as shown in the following example:

C#

```
// var is used for convenience in these queries
double average = numbers1.Average();
var concatenationQuery = numbers1.Concat(numbers2);
var largeNumbersQuery = numbers2.Where(c => c > 15);
```

Example - mixed query and method syntax

This example shows how to use method syntax on the results of a query clause. Enclose the query expression in parentheses, and then apply the dot operator and call the method. In the following example, query #7 returns a count of the numbers whose value is between 3 and 7.

C#

```
// Query #7.

// Using a query expression with method syntax
var numCount1 = (
    from num in numbers1
    where num is > 3 and < 7
    select num
).Count();

// Better: Create a new variable to store
// the method call result
IEnumerable<int> numbersQuery =
    from num in numbers1
    where num is > 3 and < 7
    select num;

var numCount2 = numbersQuery.Count();
```

Because Query #7 returns a single value and not a collection, the query executes immediately.

You can write the previous query by using implicit typing with `var`, as follows:

C#

```
var numCount = (from num in numbers...
```

You can write it in method syntax as follows:

C#

```
var numCount = numbers.Count(n => n is > 3 and < 7);
```

You can write it by using explicit typing, as follows:

C#

```
int numCount = numbers.Count(n => n is > 3 and < 7);
```

Dynamically specify predicate filters at run time

In some cases, you don't know until run time how many predicates you have to apply to source elements in the `where` clause. One way to dynamically specify multiple predicate filters is to use the Contains method, as shown in the following example. The query returns different results based on the value of `id` when the query is executed.

C#

```
int[] ids = [ 111, 114, 112 ];

var queryNames = from student in students
                 where ids.Contains(student.ID)
                 select new
                 {
                     student.LastName,
                     student.ID
                 };

foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
}

/* Output:
    Garcia: 114
    O'Donnell: 112
    Omelchenko: 111
 */

// Change the ids.
ids = [ 122, 117, 120, 115 ];

// The query will now return different results
foreach (var name in queryNames)
{
    Console.WriteLine($"{name.LastName}: {name.ID}");
```

```
}

/* Output:
    Adams: 120
    Feng: 117
    Garcia: 115
    Tucker: 122
 */
```

> **Note**
>
> This example uses the following data source and data:

C#

```
record City(string Name, long Population);
record Country(string Name, double Area, long Population, List<City> Cities);
record Product(string Name, string Category);
```

C#

```
static readonly City[] cities = [
    new City("Tokyo", 37_833_000),
    new City("Delhi", 30_290_000),
    new City("Shanghai", 27_110_000),
    new City("São Paulo", 22_043_000),
    new City("Mumbai", 20_412_000),
    new City("Beijing", 20_384_000),
    new City("Cairo", 18_772_000),
    new City("Dhaka", 17_598_000),
    new City("Osaka", 19_281_000),
    new City("New York-Newark", 18_604_000),
    new City("Karachi", 16_094_000),
    new City("Chongqing", 15_872_000),
    new City("Istanbul", 15_029_000),
    new City("Buenos Aires", 15_024_000),
    new City("Kolkata", 14_850_000),
    new City("Lagos", 14_368_000),
    new City("Kinshasa", 14_342_000),
    new City("Manila", 13_923_000),
    new City("Rio de Janeiro", 13_374_000),
    new City("Tianjin", 13_215_000)
];

static readonly Country[] countries = [
    new Country ("Vatican City", 0.44, 526, [new City("Vatican City", 826)]),
    new Country ("Monaco", 2.02, 38_000, [new City("Monte Carlo", 38_000)]),
    new Country ("Nauru", 21, 10_900, [new City("Yaren", 1_100)]),
    new Country ("Tuvalu", 26, 11_600, [new City("Funafuti", 6_200)]),
    new Country ("San Marino", 61, 33_900, [new City("San Marino", 4_500)]),
    new Country ("Liechtenstein", 160, 38_000, [new City("Vaduz", 5_200)]),
```

```
    new Country ("Marshall Islands", 181, 58_000, [new City("Majuro", 28_000)]),
    new Country ("Saint Kitts & Nevis", 261, 53_000, [new City("Basseterre",
13_000)])
];
```

You can use control flow statements, such as `if... else` or `switch`, to select among predetermined alternative queries. In the following example, `studentQuery` uses a different `where` clause if the run-time value of `oddYear` is `true` or `false`.

C#

```
void FilterByYearType(bool oddYear)
{
    IEnumerable<Student> studentQuery = oddYear
        ? (from student in students
           where student.Year is GradeLevel.FirstYear or GradeLevel.ThirdYear
           select student)
        : (from student in students
           where student.Year is GradeLevel.SecondYear or GradeLevel.FourthYear
           select student);
    var descr = oddYear ? "odd" : "even";
    Console.WriteLine($"The following students are at an {descr} year level:");
    foreach (Student name in studentQuery)
    {
        Console.WriteLine($"{name.LastName}: {name.ID}");
    }
}

FilterByYearType(true);

/* Output:
    The following students are at an odd year level:
    Fakhouri: 116
    Feng: 117
    Garcia: 115
    Mortensen: 113
    Tucker: 119
    Tucker: 122
 */

FilterByYearType(false);

/* Output:
    The following students are at an even year level:
    Adams: 120
    Garcia: 114
    Garcia: 118
    O'Donnell: 112
    Omelchenko: 111
    Zabokritski: 121
 */
```

Handle null values in query expressions

This example shows how to handle possible null values in source collections. An object collection such as an IEnumerable<T> can contain elements whose value is null. If a source collection is `null` or contains an element whose value is `null`, and your query doesn't handle `null` values, a NullReferenceException is thrown when you execute the query.

The following example uses these types and static data arrays:

C#

```
record Product(string Name, int CategoryID);
record Category(string Name, int ID);
```

C#

```
static Category?[] categories =
[
    new ("brass", 1),
    null,
    new ("winds", 2),
    default,
    new ("percussion", 3)
];

static Product?[] products =
[
    new Product("Trumpet", 1),
    new Product("Trombone", 1),
    new Product("French Horn", 1),
    null,
    new Product("Clarinet", 2),
    new Product("Flute", 2),
    null,
    new Product("Cymbal", 3),
    new Product("Drum", 3)
];
```

You can code defensively to avoid a null reference exception as shown in the following example:

C#

```
var query1 = from c in categories
             where c != null
             join p in products on c.ID equals p?.CategoryID
             select new
             {
                 Category = c.Name,
```

```
                Name = p.Name
            };
```

In the previous example, the `where` clause filters out all null elements in the categories sequence. This technique is independent of the null check in the join clause. The conditional expression with null in this example works because `Products.CategoryID` is of type `int?`, which is shorthand for `Nullable<int>`.

In a join clause, if only one of the comparison keys is a nullable value type, you can cast the other to a nullable value type in the query expression. In the following example, assume that `EmployeeID` is a column that contains values of type `int?`:

C#

```
var query =
    from o in db.Orders
    join e in db.Employees
        on o.EmployeeID equals (int?)e.EmployeeID
    select new { o.OrderID, e.FirstName };
```

In each of the examples, the `equals` query keyword is used. You can also use pattern matching, which includes patterns for `is null` and `is not null`. These patterns aren't recommended in LINQ queries because query providers might not interpret the new C# syntax correctly. A query provider is a library that translates C# query expressions into a native data format, such as Entity Framework Core. Query providers implement the System.Linq.IQueryProvider interface to create data sources that implement the System.Linq.IQueryable<T> interface.

Handle exceptions in query expressions

You can call any method in the context of a query expression. Don't call any method in a query expression that can create a side effect such as modifying the contents of the data source or throwing an exception. This example shows how to avoid raising exceptions when you call methods in a query expression without violating the general .NET guidelines on exception handling. Those guidelines state that it's acceptable to catch a specific exception when you understand why it's thrown in a given context. For more information, see Best Practices for Exceptions.

The final example shows how to handle those cases when you must throw an exception during execution of a query.

The following example shows how to move exception handling code outside a query expression. You can only refactor this way when the method doesn't depend on any variables local to the query. It's easier to deal with exceptions outside of the query expression.

C#

```
// A data source that is very likely to throw an exception!
IEnumerable<int> GetData() => throw new InvalidOperationException();

// DO THIS with a datasource that might
// throw an exception.
IEnumerable<int>? dataSource = null;
try
{
    dataSource = GetData();
}
catch (InvalidOperationException)
{
    Console.WriteLine("Invalid operation");
}

if (dataSource is not null)
{
    // If we get here, it is safe to proceed.
    var query = from i in dataSource
                select i * i;

    foreach (var i in query)
    {
        Console.WriteLine(i.ToString());
    }
}
```

In the `catch (InvalidOperationException)` block in the preceding example, handle (or don't handle) the exception in the way that's appropriate for your application.

In some cases, the best response to an exception that's thrown from within a query might be to stop the query execution immediately. The following example shows how to handle exceptions that might be thrown from inside a query body. Assume that `SomeMethodThatMightThrow` can potentially cause an exception that requires the query execution to stop.

The `try` block encloses the `foreach` loop, and not the query itself. The `foreach` loop is the point at which the query is executed. Run-time exceptions are thrown when the query is executed. Therefore, handle them in the `foreach` loop.

C#

```
// Not very useful as a general purpose method.
string SomeMethodThatMightThrow(string s) =>
    s[4] == 'C' ?
        throw new InvalidOperationException() :
        $"""C:\newFolder\{s}""";

// Data source.
string[] files = ["fileA.txt", "fileB.txt", "fileC.txt"];
```

```
// Demonstration query that throws.
var exceptionDemoQuery = from file in files
                         let n = SomeMethodThatMightThrow(file)
                         select n;

try
{
    foreach (var item in exceptionDemoQuery)
    {
        Console.WriteLine($"Processing {item}");
    }
}
catch (InvalidOperationException e)
{
    Console.WriteLine(e.Message);
}

/* Output:
    Processing C:\newFolder\fileA.txt
    Processing C:\newFolder\fileB.txt
    Operation is not valid due to the current state of the object.
 */
```

Catch whatever exception you expect to raise, and do any necessary cleanup in a `finally` block.

See also

- where clause
- Querying based on runtime state
- Nullable<T>
- Nullable value types

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 12/01/2025

Type Relationships in LINQ Query Operations (C#)

Article • 12/15/2023

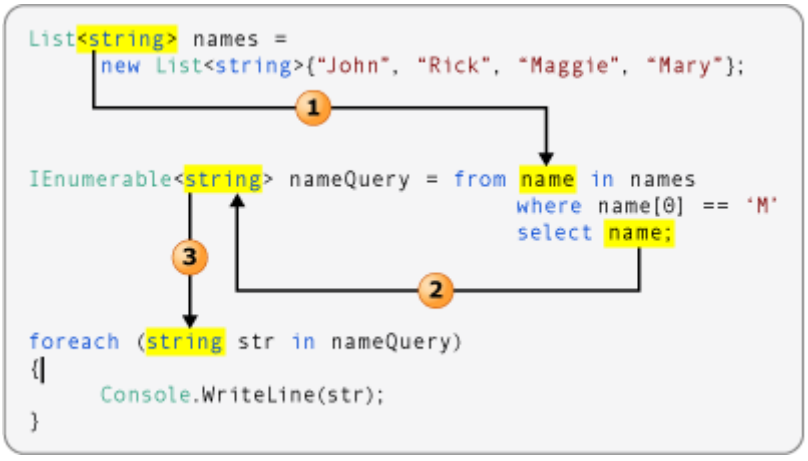
To write queries effectively, you should understand how types of the variables in a complete query operation all relate to each other. If you understand these relationships you will more easily comprehend the LINQ samples and code examples in the documentation. Furthermore, you will understand what occurs when variables are implicitly typed by using var.

LINQ query operations are strongly typed in the data source, in the query itself, and in the query execution. The type of the variables in the query must be compatible with the type of the elements in the data source and with the type of the iteration variable in the foreach statement. This strong typing guarantees that type errors are caught at compile time when they can be corrected before users encounter them.

In order to demonstrate these type relationships, most of the examples that follow use explicit typing for all variables. The last example shows how the same principles apply even when you use implicit typing by using `var`.

Queries that do not Transform the Source Data

The following illustration shows a LINQ to Objects query operation that performs no transformations on the data. The source contains a sequence of strings and the query output is also a sequence of strings.

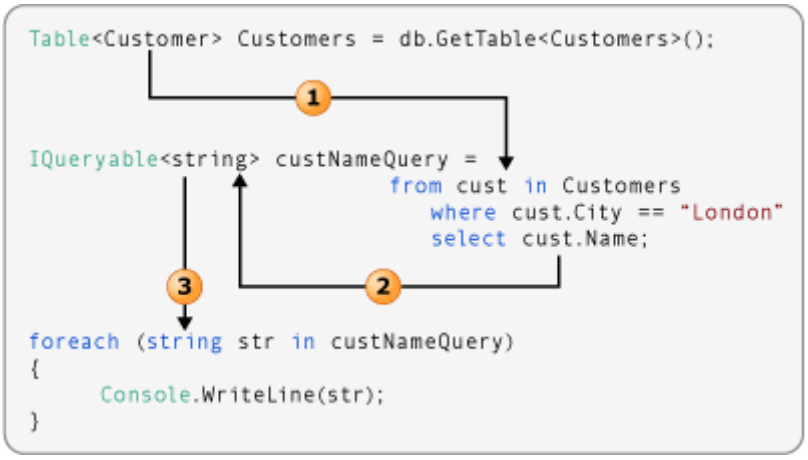


1. The type argument of the data source determines the type of the range variable.
2. The type of the object that is selected determines the type of the query variable. Here `name` is a string. Therefore, the query variable is an `IEnumerable<string>`.

3. The query variable is iterated over in the `foreach` statement. Because the query variable is a sequence of strings, the iteration variable is also a string.

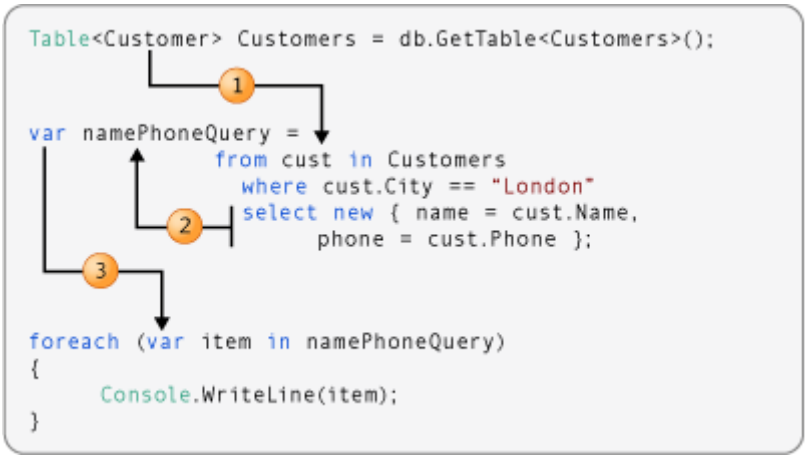
Queries that Transform the Source Data

The following illustration shows a LINQ to SQL query operation that performs a simple transformation on the data. The query takes a sequence of `Customer` objects as input, and selects only the `Name` property in the result. Because `Name` is a string, the query produces a sequence of strings as output.



1. The type argument of the data source determines the type of the range variable.
2. The `select` statement returns the `Name` property instead of the complete `Customer` object. Because `Name` is a string, the type argument of `custNameQuery` is `string`, not `Customer`.
3. Because `custNameQuery` is a sequence of strings, the `foreach` loop's iteration variable must also be a `string`.

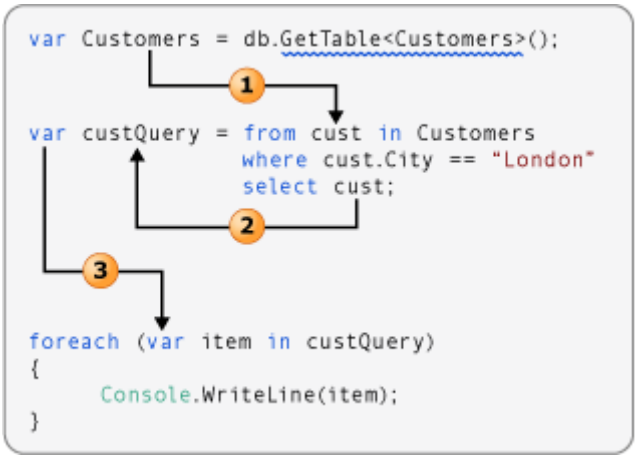
The following illustration shows a slightly more complex transformation. The `select` statement returns an anonymous type that captures just two members of the original `Customer` object.



1. The type argument of the data source is always the type of the range variable in the query.
2. Because the `select` statement produces an anonymous type, the query variable must be implicitly typed by using `var`.
3. Because the type of the query variable is implicit, the iteration variable in the `foreach` loop must also be implicit.

Letting the compiler infer type information

Although you should understand the type relationships in a query operation, you have the option to let the compiler do all the work for you. The keyword var can be used for any local variable in a query operation. The following illustration is similar to example number 2 that was discussed earlier. However, the compiler supplies the strong type for each variable in the query operation.



LINQ and generic types (C#)

LINQ queries are based on generic types. You do not need an in-depth knowledge of generics before you can start writing queries. However, you may want to understand two basic concepts:

1. When you create an instance of a generic collection class such as List<T>, you replace the "T" with the type of objects that the list will hold. For example, a list of strings is expressed as `List<string>`, and a list of `Customer` objects is expressed as `List<Customer>`. A generic list is strongly typed and provides many benefits over collections that store their elements as Object. If you try to add a `Customer` to a `List<string>`, you will get an error at compile time. It is easy to use generic collections because you do not have to perform run-time type-casting.
2. IEnumerable<T> is the interface that enables generic collection classes to be enumerated by using the `foreach` statement. Generic collection classes support

IEnumerable<T> just as non-generic collection classes such as ArrayList support IEnumerable.

For more information about generics, see Generics.

IEnumerable<T> variables in LINQ queries

LINQ query variables are typed as IEnumerable<T> or a derived type such as IQueryable<T>. When you see a query variable that is typed as `IEnumerable<Customer>`, it just means that the query, when it is executed, will produce a sequence of zero or more `Customer` objects.

C#

```
IEnumerable<Customer> customerQuery = from cust in customers
                                      where cust.City == "London"
                                      select cust;

foreach (Customer customer in customerQuery)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

Letting the compiler handle generic type declarations

If you prefer, you can avoid generic syntax by using the var keyword. The `var` keyword instructs the compiler to infer the type of a query variable by looking at the data source specified in the `from` clause. The following example produces the same compiled code as the previous example:

C#

```
var customerQuery2 = from cust in customers
                     where cust.City == "London"
                     select cust;

foreach(var customer in customerQuery2)
{
    Console.WriteLine($"{customer.LastName}, {customer.FirstName}");
}
```

The `var` keyword is useful when the type of the variable is obvious or when it is not that important to explicitly specify nested generic types such as those that are produced by group queries. In general, we recommend that if you use `var`, realize that it can make your code more difficult for others to read. For more information, see Implicitly Typed Local Variables.

C# features that support LINQ

Query expressions

Query expressions use a declarative syntax similar to SQL or XQuery to query over System.Collections.Generic.IEnumerable<T> collections. At compile time, the compiler converts query syntax to method calls to a LINQ provider's implementation of the standard query methods. Applications control the standard query operators that are in scope by specifying the appropriate namespace with a using directive. The following query expression takes an array of strings, groups them according to the first character in the string, and orders the groups.

C#

```
var query = from str in stringArray
            group str by str[0] into stringGroup
            orderby stringGroup.Key
            select stringGroup;
```

Implicitly typed variables (var)

You can use the var modifier to instruct the compiler to infer and assign the type, as shown here:

C#

```
var number = 5;
var name = "Virginia";
var query = from str in stringArray
            where str[0] == 'm'
            select str;
```

Variables declared as `var` are strongly typed, just like variables whose type you specify explicitly. Using `var` makes it possible to create anonymous types, but only for local variables. For more information, see Implicitly Typed Local Variables.

Object and collection initializers

Object and collection initializers make it possible to initialize objects without explicitly calling a constructor for the object. You typically use initializers in query expressions when they project the source data into a new data type. Assuming a class named `Customer` with public `Name` and `Phone` properties, you can use the object initializer as in the following code:

C#

```
var cust = new Customer { Name = "Mike", Phone = "555-1212" };
```

Continuing with your `Customer` class, assume that there's a data source called `IncomingOrders`, and that for each order with a large `OrderSize`, you want to create a new `Customer` based off of that order. You can execute a LINQ query on this data source and use object initialization to fill a collection:

C#

```
var newLargeOrderCustomers = from o in IncomingOrders
                             where o.OrderSize > 5
                             select new Customer { Name = o.Name, Phone = o.Phone };
```

The data source might have more properties defined than the `Customer` class such as `OrderSize`, but with object initialization, the data returned from the query is molded into the desired data type; you choose the data that's relevant to your class. As a result, you now have an System.Collections.Generic.IEnumerable<T> filled with the new `Customer`s you wanted. You can also write the preceding example in LINQ's method syntax:

C#

```
var newLargeOrderCustomers = IncomingOrders.Where(x => x.OrderSize > 5).Select(y =>
new Customer { Name = y.Name, Phone = y.Phone });
```

Beginning with C# 12, you can use a collection expression to initialize a collection.

For more information, see:

- Object and Collection Initializers
- Query Expression Syntax for Standard Query Operators

Anonymous types

The compiler constructs an anonymous type. Only the compiler can access the type name. Anonymous types provide a convenient way to group a set of properties temporarily in a query result without having to define a separate named type. You initialize anonymous types with a new expression and an object initializer, as shown here:

C#

```
select new {name = cust.Name, phone = cust.Phone};
```

Beginning with C# 7, you can use tuples to create unnamed types.

Extension members

An extension member is a static member of a static class associated with a type called the *receiver type*. You can call an extension member as if it were a member of the receiver type. This feature enables you to "add" new members to existing types without actually modifying them. The standard query operators are a set of extension methods that provide LINQ query functionality for any type that implements IEnumerable<T>.

Lambda expressions

A lambda expression is an inline function that uses the `=>` operator to separate input parameters from the function body and can be converted at compile time to a delegate or an expression tree. In LINQ programming, you encounter lambda expressions when you make direct method calls to the standard query operators.

Expressions as data

Query objects are composable, meaning that you can return a query from a method. Objects that represent queries don't store the resulting collection, but rather the steps to produce the results when needed. The advantage of returning query objects from methods is that you can further compose or modify them. Therefore, any return value or `out` parameter of a method that returns a query must also have that type. If a method materializes a query into a concrete List<T> or Array type, it returns the query results instead of the query itself. You can still compose or modify a query variable that's returned from a method.

In the following example, the first method `QueryMethod1` returns a query as a return value, and the second method `QueryMethod2` returns a query as an `out` parameter (`returnQ` in the example). In both cases, it's a query that's returned, not query results.

C#

```
IEnumerable<string> QueryMethod1(int[] ints) =>
    from i in ints
    where i > 4
    select i.ToString();

void QueryMethod2(int[] ints, out IEnumerable<string> returnQ) =>
    returnQ = from i in ints
              where i < 4
              select i.ToString();
```

```
int[] nums = [ 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 ];

var myQuery1 = QueryMethod1(nums);
```

The following `foreach` loop executes query `myQuery1`.

C#

```
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

Rest the mouse pointer over `myQuery1` to see its type.

You can also execute the query returned from `QueryMethod1` directly, without using `myQuery1`.

C#

```
foreach (var s in QueryMethod1(nums))
{
    Console.WriteLine(s);
}
```

Rest the mouse pointer over the call to `QueryMethod1` to see its return type.

`QueryMethod2` returns a query as the value of its `out` parameter:

C#

```
QueryMethod2(nums, out IEnumerable<string> myQuery2);

// Execute the returned query.
foreach (var s in myQuery2)
{
    Console.WriteLine(s);
}
```

You can modify a query by using query composition. In this case, the previous query object is used to create a new query object. This new object returns different results than the original query object.

C#

```
myQuery1 = from item in myQuery1
           orderby item descending
           select item;
```

```
// Execute the modified query.
Console.WriteLine("\nResults of executing modified myQuery1:");
foreach (var s in myQuery1)
{
    Console.WriteLine(s);
}
```

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 12/03/2025

Tutorial: Write queries in C# using language integrated query (LINQ)

Article • 04/25/2024

In this tutorial, you create a data source and write several LINQ queries. You can experiment with the query expressions and see the differences in the results. This walkthrough demonstrates the C# language features that are used to write LINQ query expressions. You can follow along and build the app and experiment with the queries yourself. This article assumes you've installed the latest .NET SDK. If not, go to the .NET Downloads page and install the latest version on your machine.

First, create the application. From the console, type the following command:

.NET CLI

```
dotnet new console -o WalkthroughWritingLinqQueries
```

Or, if you prefer Visual Studio, create a new console application named *WalkthroughWritingLinqQueries*.

Create an in-memory data source

The first step is to create a data source for your queries. The data source for the queries is a simple list of `Student` records. Each `Student` record has a first name, family name, and an array of integers that represents their test scores in the class. Add a new file named *students.cs*, and copy the following code into that file:

C#

```
namespace WalkthroughWritingLinqQueries;

public record Student(string First, string Last, int ID, int[] Scores);
```

Note the following characteristics:

- The `Student` record consists of automatically implemented properties.
- Each student in the list is initialized with the primary constructor.
- The sequence of scores for each student is initialized with a primary constructor.

Next, create a sequence of `Student` records that serves as the source of this query. Open *Program.cs*, and remove the following boilerplate code:

C#

```
// See https://aka.ms/new-console-template for more information
Console.WriteLine("Hello, World!");
```

Replace it with the following code that creates a sequence of `Student` records:

C#

```
using WalkthroughWritingLinqQueries;

// Create a data source by using a collection initializer.
IEnumerable<Student> students =
[
    new Student(First: "Svetlana", Last: "Omelchenko", ID: 111, Scores: [97,
92, 81, 60]),
    new Student(First: "Claire",   Last: "O'Donnell",  ID: 112, Scores: [75,
84, 91, 39]),
    new Student(First: "Sven",     Last: "Mortensen",  ID: 113, Scores: [88,
94, 65, 91]),
    new Student(First: "Cesar",    Last: "Garcia",     ID: 114, Scores: [97,
89, 85, 82]),
    new Student(First: "Debra",    Last: "Garcia",     ID: 115, Scores: [35,
72, 91, 70]),
    new Student(First: "Fadi",     Last: "Fakhouri",   ID: 116, Scores: [99,
86, 90, 94]),
    new Student(First: "Hanying",  Last: "Feng",       ID: 117, Scores: [93,
92, 80, 87]),
    new Student(First: "Hugo",     Last: "Garcia",     ID: 118, Scores: [92,
90, 83, 78]),

    new Student("Lance",   "Tucker",      119, [68, 79, 88, 92]),
    new Student("Terry",   "Adams",       120, [99, 82, 81, 79]),
    new Student("Eugene",  "Zabokritski", 121, [96, 85, 91, 60]),
    new Student("Michael", "Tucker",      122, [94, 92, 91, 91])
];
```

- The sequence of students is initialized with a collection expression.
- The `Student` record type holds the static list of all students.
- Some of the constructor calls use named arguments to clarify which argument matches which constructor parameter.

Try adding a few more students with different test scores to the list of students to get more familiar with the code so far.

Create the query

Next, you create your first query. Your query, when you execute it, produces a list of all students whose score on the first test was greater than 90. Because the whole `Student` object is selected, the type of the query is `IEnumerable<Student>`. Although the code could also use implicit typing by using the var keyword, explicit typing is used to clearly illustrate results. (For more information about `var`, see Implicitly Typed Local Variables.) Add the following code to *Program.cs*, after the code that creates the sequence of students:

C#

```
// Create the query.
// The first line could also be written as "var studentQuery ="
IEnumerable<Student> studentQuery =
    from student in students
    where student.Scores[0] > 90
    select student;
```

The query's range variable, `student`, serves as a reference to each `Student` in the source, providing member access for each object.

Run the query

Now write the `foreach` loop that causes the query to execute. Each element in the returned sequence is accessed through the iteration variable in the `foreach` loop. The type of this variable is `Student`, and the type of the query variable is compatible, `IEnumerable<Student>`. After you added the following code, build and run the application to see the results in the **Console** window.

C#

```
// Execute the query.
// var could be used here also.
foreach (Student student in studentQuery)
{
    Console.WriteLine($"{student.Last}, {student.First}");
}

// Output:
// Omelchenko, Svetlana
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
```

```
// Zabokritski, Eugene
// Tucker, Michael
```

To further refine the query, you can combine multiple Boolean conditions in the `where` clause. The following code adds a condition so that the query returns those students whose first score was over 90 and whose last score was less than 80. The `where` clause should resemble the following code.

C#

```
where student.Scores[0] > 90 && student.Scores[3] < 80
```

Try the preceding `where` clause, or experiment yourself with other filter conditions. For more information, see where clause.

Order the query results

It's easier to scan the results if they are in some kind of order. You can order the returned sequence by any accessible field in the source elements. For example, the following `orderby` clause orders the results in alphabetical order from A to Z according to the family name of each student. Add the following `orderby` clause to your query, right after the `where` statement and before the `select` statement:

C#

```
orderby student.Last ascending
```

Now change the `orderby` clause so that it orders the results in reverse order according to the score on the first test, from the highest score to the lowest score.

C#

```
orderby student.Scores[0] descending
```

Change the `WriteLine` format string so that you can see the scores:

C#

```
Console.WriteLine($"{student.Last}, {student.First} {student.Scores[0]}");
```

For more information, see orderby clause.

Group the results

Grouping is a powerful capability in query expressions. A query with a group clause produces a sequence of groups, and each group itself contains a `Key` and a sequence that consists of all the members of that group. The following new query groups the students by using the first letter of their family name as the key.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];
```

The type of the query changed. It now produces a sequence of groups that have a `char` type as a key, and a sequence of `Student` objects. The code in the `foreach` execution loop also must change:

C#

```
foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
// Output:
// O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
// M
//   Mortensen, Sven
// G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
// F
//   Fakhouri, Fadi
//   Feng, Hanying
// T
//   Tucker, Lance
//   Tucker, Michael
// A
//   Adams, Terry
// Z
//   Zabokritski, Eugene
```

Run the application and view the results in the **Console** window. For more information, see group clause.

Explicitly coding `IEnumerables` of `IGroupings` can quickly become tedious. Write the same query and `foreach` loop much more conveniently by using `var`. The `var` keyword doesn't change the types of your objects; it just instructs the compiler to infer the types. Change the type of `studentQuery` and the iteration variable `group` to `var` and rerun the query. In the inner `foreach` loop, the iteration variable is still typed as `Student`, and the query works as before. Change the `student` iteration variable to `var` and run the query again. You see that you get exactly the same results.

C#

```
IEnumerable<IGrouping<char, Student>> studentQuery =
    from student in students
    group student by student.Last[0];

foreach (IGrouping<char, Student> studentGroup in studentQuery)
{
    Console.WriteLine(studentGroup.Key);
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}
```

For more information about `var`, see Implicitly Typed Local Variables.

Order the groups by their key value

The groups in the previous query aren't in alphabetical order. You can provide an `orderby` clause after the `group` clause. But to use an `orderby` clause, you first need an identifier that serves as a reference to the groups created by the `group` clause. You provide the identifier by using the `into` keyword, as follows:

C#

```
var studentQuery4 =
    from student in students
    group student by student.Last[0] into studentGroup
    orderby studentGroup.Key
    select studentGroup;

foreach (var groupOfStudents in studentQuery4)
{
    Console.WriteLine(groupOfStudents.Key);
```

```
    foreach (var student in groupOfStudents)
    {
        Console.WriteLine($"   {student.Last}, {student.First}");
    }
}

// Output:
//A
//   Adams, Terry
//F
//   Fakhouri, Fadi
//   Feng, Hanying
//G
//   Garcia, Cesar
//   Garcia, Debra
//   Garcia, Hugo
//M
//   Mortensen, Sven
//O
//   Omelchenko, Svetlana
//   O'Donnell, Claire
//T
//   Tucker, Lance
//   Tucker, Michael
//Z
//   Zabokritski, Eugene
```

Run this query, and the groups are now sorted in alphabetical order.

You can use the `let` keyword to introduce an identifier for any expression result in the query expression. This identifier can be a convenience, as in the following example. It can also enhance performance by storing the results of an expression so that it doesn't have to be calculated multiple times.

C#

```
// This query returns those students whose
// first test score was higher than their
// average score.
var studentQuery5 =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where totalScore / 4 < student.Scores[0]
    select $"{student.Last}, {student.First}";

foreach (string s in studentQuery5)
{
    Console.WriteLine(s);
}

// Output:
```

```
// Omelchenko, Svetlana
// O'Donnell, Claire
// Mortensen, Sven
// Garcia, Cesar
// Fakhouri, Fadi
// Feng, Hanying
// Garcia, Hugo
// Adams, Terry
// Zabokritski, Eugene
// Tucker, Michael
```

For more information, see the article on the let clause.

Use method syntax in a query expression

As described in Query Syntax and Method Syntax in LINQ, some query operations can only be expressed by using method syntax. The following code calculates the total score for each `Student` in the source sequence, and then calls the `Average()` method on the results of that query to calculate the average score of the class.

C#

```
var studentQuery =
    from student in students
    let totalScore = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    select totalScore;

double averageScore = studentQuery.Average();
Console.WriteLine($"Class average score = {averageScore}");

// Output:
// Class average score = 334.166666666667
```

To transform or project in the select clause

It's common for a query to produce a sequence whose elements differ from the elements in the source sequences. Delete or comment out your previous query and execution loop, and replace it with the following code. The query returns a sequence of strings (not `Students`), and this fact is reflected in the `foreach` loop.

C#

```
IEnumerable<string> studentQuery =
    from student in students
    where student.Last == "Garcia"
```

```
    select student.First;

Console.WriteLine("The Garcias in the class are:");
foreach (string s in studentQuery)
{
    Console.WriteLine(s);
}

// Output:
// The Garcias in the class are:
// Cesar
// Debra
// Hugo
```

Code earlier in this walkthrough indicated that the average class score is approximately 334. To produce a sequence of `Students` whose total score is greater than the class average, together with their `Student ID`, you can use an anonymous type in the `select` statement:

C#

```
var aboveAverageQuery =
    from student in students
    let x = student.Scores[0] + student.Scores[1] +
        student.Scores[2] + student.Scores[3]
    where x > averageScore
    select new { id = student.ID, score = x };

foreach (var item in aboveAverageQuery)
{
    Console.WriteLine("Student ID: {0}, Score: {1}", item.id, item.score);
}

// Output:
// Student ID: 113, Score: 338
// Student ID: 114, Score: 353
// Student ID: 116, Score: 369
// Student ID: 117, Score: 352
// Student ID: 118, Score: 343
// Student ID: 120, Score: 341
// Student ID: 122, Score: 368
```

Standard query operators overview

The *standard query operators* are the keywords and methods that form the LINQ pattern. The C# language defines LINQ query keywords that you use for the most common query expression. The compiler translates expressions using these keywords to the equivalent method calls. The two forms are synonymous. Other methods that are part of the System.Linq namespace don't have equivalent query keywords. In those cases, you must use the method syntax. This section covers all the query operator keywords. The runtime and other NuGet packages add more methods designed to work with LINQ queries each release. The most common methods, including those that have query keyword equivalents, are covered in this section. For the full list of query methods supported by the .NET Runtime, see the System.Linq.Enumerable API documentation. In addition to the methods covered here, this class contains methods for concatenating data sources, computing a single value from a data source, such as a sum, average, or other value.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

Most of these methods operate on sequences, where a sequence is an object whose type implements the IEnumerable<T> interface or the IQueryable<T> interface. The standard query operators provide query capabilities including filtering, projection, aggregation, sorting, and more. The methods that make up each set are extension members defined in the Enumerable and Queryable classes, respectively. They're defined as *extension members* where the receiver type is either the `IEnumerable<T>` or `IQueryable<T>` type that they operate on.

The distinction between IEnumerable<T> and IQueryable<T> sequences determines how the query is executed at runtime.

For `IEnumerable<T>`, the returned enumerable object captures the arguments that were passed to the method. When that object is enumerated, the logic of the query operator is employed and the query results are returned.

For `IQueryable<T>`, the query is translated into an expression tree. The expression tree can be translated to a native query when the data source can optimize the query. Libraries such as Entity Framework translate LINQ queries into native SQL queries that execute at the database.

The following code example demonstrates how you can use the standard query operators to obtain information about a sequence.

C#

```
string sentence = "the quick brown fox jumps over the lazy dog";
// Split the string into individual words to create a collection.
string[] words = sentence.Split(' ');

// Using query expression syntax.
var query = from word in words
            group word.ToUpper() by word.Length into gr
            orderby gr.Key
            select new { Length = gr.Key, Words = gr };

// Using method-based query syntax.
var query2 = words.
    GroupBy(w => w.Length, w => w.ToUpper()).
    Select(g => new { Length = g.Key, Words = g }).
    OrderBy(o => o.Length);

foreach (var obj in query)
{
    Console.WriteLine($"Words of length {obj.Length}:");
    foreach (string word in obj.Words)
        Console.WriteLine(word);
}

// This code example produces the following output:
//
// Words of length 3:
// THE
// FOX
// THE
// DOG
// Words of length 4:
// OVER
// LAZY
// Words of length 5:
// QUICK
// BROWN
// JUMPS
```

Where possible, the queries in this section use a sequence of words or numbers as the input source. For queries where more complicated relationships between objects are used, the following sources that model a school are used:

C#

```
public enum GradeLevel
{
    FirstYear = 1,
```

```
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name and a reference to a `Teacher` who serves as the department head.

You can find the data set in the source repo.

Types of query operators

The standard query operators differ in the timing of their execution, depending on whether they return a singleton value or a sequence of values. Those methods that return a singleton value (such as Average and Sum) execute immediately. Methods that return a sequence defer the query execution and return an enumerable object. You can use the output sequence of one query as the input sequence to another query. You chain query methods together in one query, which enables queries to become arbitrarily complex.

Query operators

In a LINQ query, the first step is to specify the data source. In a LINQ query, the `from` clause comes first in order to introduce the data source (`students`) and the *range variable* (`student`).

C#

```
//queryAllStudents is an IEnumerable<Student>
var queryAllStudents = from student in students
                       select student;
```

The range variable is like the iteration variable in a `foreach` loop except that no actual iteration occurs in a query expression. When the query is executed, the range variable serves as a reference to each successive element in `students`. Because the compiler can infer the type of `student`, you don't have to specify it explicitly. You can introduce more range variables in a `let` clause. For more information, see let clause.

> **Note**
>
> For non-generic data sources such as ArrayList, you must explicitly type the range variable. For more information, see How to query an ArrayList with LINQ (C#) and from clause.

Once you obtain a data source, you can perform any number of operations on that data source:

- Filter data using the `where` keyword.
- Order data using the `orderby` and optionally `descending` keywords.
- Group data using the `group` and optionally `into` keywords.
- Join data using the `join` keyword.
- Project data using the `select` keyword.

Query expression syntax table

The following table lists the standard query operators that have equivalent query expression clauses.

Expand table

Method	C# query expression syntax
Cast	Use an explicitly typed range variable: `from int i in numbers` (For more information, see from clause.)

GroupBy	`group … by …`
	
	-or-
	
	`group … by … into …`
	
	(For more information, see group clause.)

GroupJoin<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,IEnumerable<TInner>, TResult>)	`join … in … on … equals … into …` (For more information, see join clause.)
Join<TOuter,TInner,TKey,TResult>(IEnumerable<TOuter>, IEnumerable<TInner>, Func<TOuter,TKey>, Func<TInner,TKey>, Func<TOuter,TInner,TResult>)	`join … in … on … equals …` (For more information, see join clause.)
OrderBy<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby` (For more information, see orderby clause.)
OrderByDescending<TSource,TKey>(IEnumerable<TSource>, Func<TSource,TKey>)	`orderby … descending` (For more information, see orderby clause.)

Method	C# query expression syntax
Select	`select` (For more information, see select clause.)
SelectMany	Multiple `from` clauses. (For more information, see from clause.)
ThenBy<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, …` (For more information, see orderby clause.)
ThenByDescending<TSource,TKey>(IOrderedEnumerable<TSource>, Func<TSource,TKey>)	`orderby …, … descending` (For more information, see orderby clause.)
Where	`where` (For more information, see where clause.)

Data transformations with LINQ

Language-Integrated Query (LINQ) isn't only about retrieving data. It's also a powerful tool for transforming data. By using a LINQ query, you can use a source sequence as input and modify it in many ways to create a new output sequence. You can modify the sequence itself without modifying the elements themselves by sorting and grouping. But perhaps the most powerful feature of LINQ queries is the ability to create new types. The select clause creates an output element from an input element. You use it to transform an input element into an output element:

- Merge multiple input sequences into a single output sequence that has a new type.

- Create output sequences whose elements consist of only one or several properties of each element in the source sequence.
- Create output sequences whose elements consist of the results of operations performed on the source data.
- Create output sequences in a different format. For example, you can transform data from SQL rows or text files into XML.

You combine these transformations in various ways in the same query. Furthermore, you can use the output sequence of one query as the input sequence for a new query. The following example transforms objects in an in-memory data structure into XML elements.

C#

```
// Create the query.
var studentsToXML = new XElement("Root",
    from student in students
    let scores = string.Join(",", student.Scores)
    select new XElement("student",
                new XElement("First", student.FirstName),
                new XElement("Last", student.LastName),
                new XElement("Scores", scores)
            ) // end "student"
        ); // end "Root"

// Execute the query.
Console.WriteLine(studentsToXML);
```

The code produces the following XML output:

XML

```
<Root>
  <student>
    <First>Svetlana</First>
    <Last>Omelchenko</Last>
    <Scores>97,90,73,54</Scores>
  </student>
  <student>
    <First>Claire</First>
    <Last>O'Donnell</Last>
    <Scores>56,78,95,95</Scores>
  </student>
  ...
  <student>
    <First>Max</First>
    <Last>Lindgren</Last>
    <Scores>86,88,96,63</Scores>
  </student>
  <student>
```

```
    <First>Arina</First>
    <Last>Ivanova</Last>
    <Scores>93,63,70,80</Scores>
  </student>
</Root>
```

For more information, see Creating XML Trees in C# (LINQ to XML).

You can use the results of one query as the data source for a subsequent query. This example shows how to order the results of a join operation. This query creates a group join and then sorts the groups based on the `department` element, which is still in scope. Inside the anonymous type initializer, a subquery orders all the matching elements from the `students` sequence.

C#

```
var orderedQuery = from department in departments
                   join student in students on department.ID equals
student.DepartmentID into studentGroup
                   orderby department.Name
                   select new
                   {
                       DepartmentName = department.Name,
                       Students = from student in studentGroup
                                  orderby student.LastName
                                    select student
                   };

foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10} {student.FirstName,-10}");
    }
}
/* Output:
Chemistry
  Balzan     Josephine
  Fakhouri   Fadi
  Popov      Innocenty
  Seleznyova Sofiya
  Vella      Carmen
Economics
  Adams      Terry
  Adaobi     Izuchukwu
  Berggren   Jeanette
  Garcia     Cesar
  Ifeoma     Nwanneka
  Jamuike    Ifeanacho
  Larsson    Naima
  Svensson   Noel
```

```
  Ugomma     Ifunanya
Engineering
  Axelsson   Erik
  Berg       Veronika
  Engström   Nancy
  Hicks      Cassie
  Keever     Bruce
  Micallef   Nicholas
  Mortensen  Sven
  Nilsson    Erna
  Tucker     Michael
  Yermolayeva Anna
English
  Andersson  Sarah
  Feng       Hanying
  Ivanova    Arina
  Jakobsson  Jesper
  Jensen     Christiane
  Johansson  Mark
  Kolpakova  Nadezhda
  Omelchenko Svetlana
  Urquhart   Donald
Mathematics
  Frost      Gaby
  Garcia     Hugo
  Hedlund    Anna
  Kovaleva   Katerina
  Lindgren   Max
  Maslova    Evgeniya
  Olsson     Ruth
  Sammut     Maria
  Sazonova   Anastasiya
Physics
  Åkesson    Sami
  Edwards    Amy E.
  Falzon     John
  Garcia     Debra
  Hansson    Sanna
  Mattsson   Martina
  Richardson Don
  Zabokritski Eugene
*/
```

The equivalent query using method syntax is shown in the following code:

C#

```
var orderedQuery = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, studentGroup) => new
    {
        DepartmentName = department.Name,
        Students = studentGroup.OrderBy(student => student.LastName)
```

```
    })
    .OrderBy(department => department.DepartmentName);


foreach (var departmentList in orderedQuery)
{
    Console.WriteLine(departmentList.DepartmentName);
    foreach (var student in departmentList.Students)
    {
        Console.WriteLine($"  {student.LastName,-10} {student.FirstName,-10}");
    }
}
```

Although you can use an `orderby` clause with one or more of the source sequences before the join, we generally don't recommend it. Some LINQ providers might not preserve that ordering after the join. For more information, see join clause.

See also

- Enumerable
- Queryable
- select clause
- Extension members
- Query Keywords (LINQ)
- Anonymous Types

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 12/01/2025

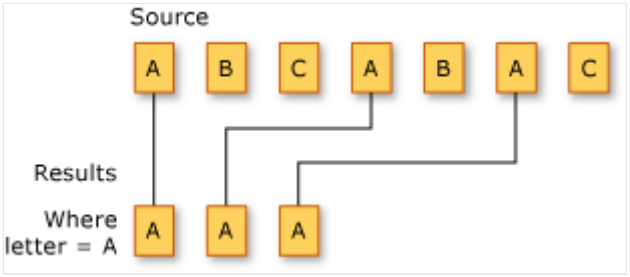
Filtering Data in C# with LINQ

08/13/2025

Filtering refers to the operation of restricting the result set to contain only those elements that satisfy a specified condition. It's also referred to as *selecting* elements that match the specified condition.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

The following illustration shows the results of filtering a sequence of characters. The predicate for the filtering operation specifies that the character must be 'A'.



The standard query operator methods that perform selection are listed in the following table:

Expand table

Method Name	Description	C# Query Expression Syntax	More Information
OfType	Selects values, depending on their ability to be cast to a specified type.	Not applicable.	Enumerable.OfType Queryable.OfType
Where	Selects values that are based on a predicate function.	`where`	Enumerable.Where Queryable.Where

The following example uses the `where` clause to filter from an array those strings that have a specific length.

> **Note**
>
> You can refer to the common data sources for this area in the Standard Query Operators Overview article.

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query = from word in words
                            where word.Length == 3
                            select word;

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/
```

The equivalent query using method syntax is shown in the following code:

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumps"];

IEnumerable<string> query =
    words.Where(word => word.Length == 3);

foreach (string str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:

    the
    fox
*/
```

See also

- System.Linq

- where clause
- How to query an assembly's metadata with Reflection (LINQ) (C#)
- How to query for files with a specified attribute or name (C#)
- How to sort or filter text data by any word or field (LINQ) (C#)

Projection operations (C#)

Article • 05/31/2024

Projection refers to the operation of transforming an object into a new form that often consists only of those properties subsequently used. By using projection, you can construct a new type that is built from each object. You can project a property and perform a mathematical function on it. You can also project the original object without changing it.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

The standard query operator methods that perform projection are listed in the following section.

Methods

Expand table

Method names	Description	C# query expression syntax	More information
Select	Projects values that are based on a transform function.	`select`	Enumerable.Select Queryable.Select
SelectMany	Projects sequences of values that are based on a transform function and then flattens them into one sequence.	Use multiple `from` clauses	Enumerable.SelectMany Queryable.SelectMany
Zip	Produces a sequence of tuples with elements from 2-3 specified sequences.	Not applicable.	Enumerable.Zip Queryable.Zip

Select

The following example uses the `select` clause to project the first letter from each string in a list of strings.

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = from word in words
            select word.Substring(0, 1);

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

The equivalent query using method syntax is shown in the following code:

C#

```
List<string> words = ["an", "apple", "a", "day"];

var query = words.Select(word => word.Substring(0, 1));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    a
    a
    a
    d
*/
```

SelectMany

The following example uses multiple `from` clauses to project each word from each string in a list of strings.

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = from phrase in phrases
            from word in phrase.Split(' ')
            select word;

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
    fox
*/
```

The equivalent query using method syntax is shown in the following code:

C#

```
List<string> phrases = ["an apple a day", "the quick brown fox"];

var query = phrases.SelectMany(phrases => phrases.Split(' '));

foreach (string s in query)
{
    Console.WriteLine(s);
}

/* This code produces the following output:

    an
    apple
    a
    day
    the
    quick
    brown
```

```
    fox
*/
```

The `SelectMany` method can also form the combination of matching every item in the first sequence with every item in the second sequence:

C#

```
var query = from number in numbers
            from letter in letters
            select (number, letter);

foreach (var item in query)
{
    Console.WriteLine(item);
}
```

The equivalent query using method syntax is shown in the following code:

C#

```
var method = numbers
    .SelectMany(number => letters,
    (number, letter) => (number, letter));

foreach (var item in method)
{
    Console.WriteLine(item);
}
```

Zip

There are several overloads for the `Zip` projection operator. All of the `Zip` methods work on sequences of two or more possibly heterogenous types. The first two overloads return tuples, with the corresponding positional type from the given sequences.

Consider the following collections:

C#

```
// An int array with 7 elements.
IEnumerable<int> numbers = [1, 2, 3, 4, 5, 6, 7];
// A char array with 6 elements.
IEnumerable<char> letters = ['A', 'B', 'C', 'D', 'E', 'F'];
```

To project these sequences together, use the Enumerable.Zip<TFirst,TSecond>(IEnumerable<TFirst>, IEnumerable<TSecond>) operator:

C#

```
foreach ((int number, char letter) in numbers.Zip(letters))
{
    Console.WriteLine($"Number: {number} zipped with letter: '{letter}'");
}
// This code produces the following output:
//     Number: 1 zipped with letter: 'A'
//     Number: 2 zipped with letter: 'B'
//     Number: 3 zipped with letter: 'C'
//     Number: 4 zipped with letter: 'D'
//     Number: 5 zipped with letter: 'E'
//     Number: 6 zipped with letter: 'F'
```

> **Important**
>
> The resulting sequence from a zip operation is never longer in length than the shortest sequence. The `numbers` and `letters` collections differ in length, and the resulting sequence omits the last element from the `numbers` collection, as it has nothing to zip with.

The second overload accepts a `third` sequence. Let's create another collection, namely `emoji`:

C#

```
// A string array with 8 elements.
IEnumerable<string> emoji = [ "😳", "🔥", "🎉", "👀", "⭐", "💜", "✔️",
"💯"];
```

To project these sequences together, use the Enumerable.Zip<TFirst,TSecond,TThird>(IEnumerable<TFirst>, IEnumerable<TSecond>, IEnumerable<TThird>) operator:

C#

```
foreach ((int number, char letter, string em) in numbers.Zip(letters,
emoji))
{
    Console.WriteLine(
        $"Number: {number} is zipped with letter: '{letter}' and emoji:
{em}");
}
// This code produces the following output:
```

```
//     Number: 1 is zipped with letter: 'A' and emoji: 😳
//     Number: 2 is zipped with letter: 'B' and emoji: 🔥
//     Number: 3 is zipped with letter: 'C' and emoji: 🎉
//     Number: 4 is zipped with letter: 'D' and emoji: 👀
//     Number: 5 is zipped with letter: 'E' and emoji: ⭐
//     Number: 6 is zipped with letter: 'F' and emoji: 💜
```

Much like the previous overload, the `Zip` method projects a tuple, but this time with three elements.

The third overload accepts a `Func<TFirst, TSecond, TResult>` argument that acts as a results selector. You can project a new resulting sequence from the sequences being zipped.

C#

```
foreach (string result in
    numbers.Zip(letters, (number, letter) => $"{number} = {letter}
({(int)letter})"))
{
    Console.WriteLine(result);
}
// This code produces the following output:
//     1 = A (65)
//     2 = B (66)
//     3 = C (67)
//     4 = D (68)
//     5 = E (69)
//     6 = F (70)
```

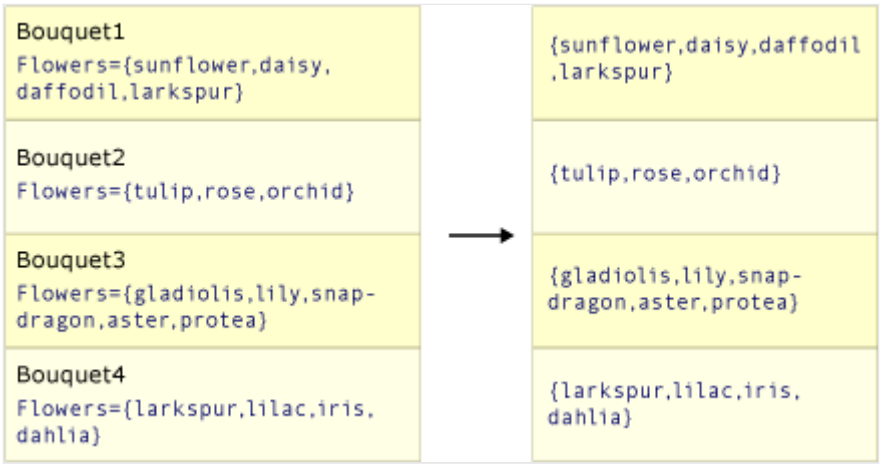
With the preceding `Zip` overload, the specified function is applied to the corresponding elements `numbers` and `letter`, producing a sequence of the `string` results.

`Select` versus `SelectMany`

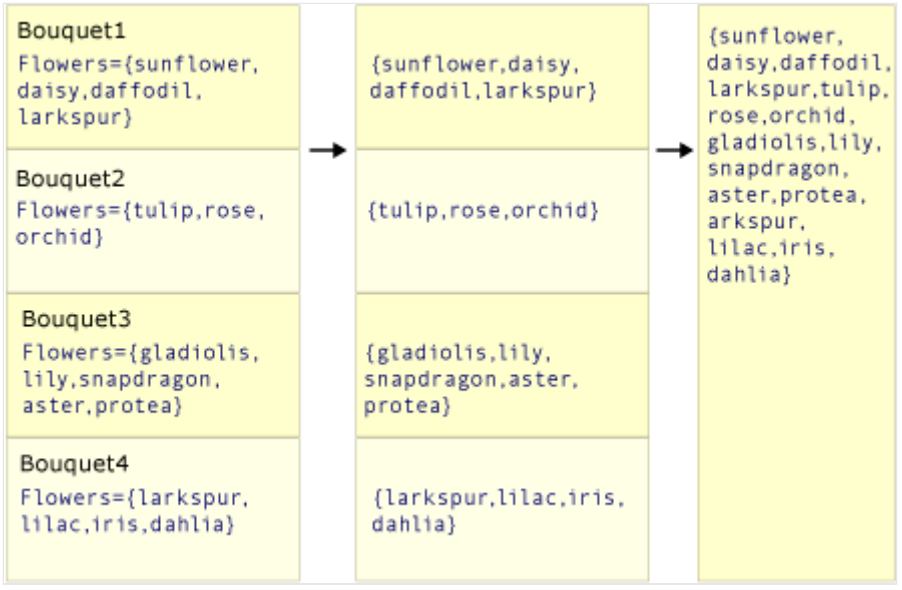
The work of both `Select` and `SelectMany` is to produce a result value (or values) from source values. `Select` produces one result value for every source value. The overall result is therefore a collection that has the same number of elements as the source collection. In contrast, `SelectMany` produces a single overall result that contains concatenated subcollections from each source value. The transform function that is passed as an argument to `SelectMany` must return an enumerable sequence of values for each source value. `SelectMany` concatenates these enumerable sequences to create one large sequence.

The following two illustrations show the conceptual difference between the actions of these two methods. In each case, assume that the selector (transform) function selects the array of flowers from each source value.

This illustration depicts how `Select` returns a collection that has the same number of elements as the source collection.



This illustration depicts how `SelectMany` concatenates the intermediate sequence of arrays into one final result value that contains each value from each intermediate array.



Code example

The following example compares the behavior of `Select` and `SelectMany`. The code creates a "bouquet" of flowers by taking the items from each list of flower names in the source collection. In the following example, the "single value" that the transform function Select<TSource,TResult>(IEnumerable<TSource>, Func<TSource,TResult>) uses is a collection of values. This example requires the extra `foreach` loop in order to enumerate each string in each subsequence.

C#

```
class Bouquet
{
    public required List<string> Flowers { get; init; }
}

static void SelectVsSelectMany()
{
    List<Bouquet> bouquets =
    [
        new Bouquet { Flowers = ["sunflower", "daisy", "daffodil",
"larkspur"] },
        new Bouquet { Flowers = ["tulip", "rose", "orchid"] },
        new Bouquet { Flowers = ["gladiolis", "lily", "snapdragon", "aster",
"protea"] },
        new Bouquet { Flowers = ["larkspur", "lilac", "iris", "dahlia"] }
    ];

    IEnumerable<List<string>> query1 = bouquets.Select(bq => bq.Flowers);

    IEnumerable<string> query2 = bouquets.SelectMany(bq => bq.Flowers);

    Console.WriteLine("Results by using Select():");
    // Note the extra foreach loop here.
    foreach (IEnumerable<string> collection in query1)
    {
        foreach (string item in collection)
        {
            Console.WriteLine(item);
        }
    }

    Console.WriteLine("\nResults by using SelectMany():");
    foreach (string item in query2)
    {
        Console.WriteLine(item);
    }
}
```

See also

- System.Linq
- select clause
- How to populate object collections from multiple sources (LINQ) (C#)
- How to split a file into many files by using groups (LINQ) (C#)

Set operations (C#)

Article • 05/31/2024

Set operations in LINQ refer to query operations that produce a result set based on the presence or absence of equivalent elements within the same or separate collections.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

Expand table

Method names	Description	C# query expression syntax	More information
`Distinct` or `DistinctBy`	Removes duplicate values from a collection.	Not applicable.	Enumerable.Distinct Enumerable.DistinctBy Queryable.Distinct Queryable.DistinctBy
`Except` or `ExceptBy`	Returns the set difference, which means the elements of one collection that don't appear in a second collection.	Not applicable.	Enumerable.Except Enumerable.ExceptBy Queryable.Except Queryable.ExceptBy
`Intersect` or `IntersectBy`	Returns the set intersection, which means elements that appear in each of two collections.	Not applicable.	Enumerable.Intersect Enumerable.IntersectBy Queryable.Intersect Queryable.IntersectBy
`Union` or `UnionBy`	Returns the set union, which means unique elements that appear in either of two collections.	Not applicable.	Enumerable.Union Enumerable.UnionBy Queryable.Union Queryable.UnionBy

`Distinct` and `DistinctBy`

The following example depicts the behavior of the Enumerable.Distinct method on a sequence of strings. The returned sequence contains the unique elements from the input sequence.



C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

IEnumerable<string> query = from word in words.Distinct()
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
 */
```

The DistinctBy is an alternative approach to `Distinct` that takes a `keySelector`. The `keySelector` is used as the comparative discriminator of the source type. In the following code, words are discriminated based on their `Length`, and the first word of each length is displayed:

C#

```
string[] words = ["the", "quick", "brown", "fox", "jumped", "over", "the",
"lazy", "dog"];

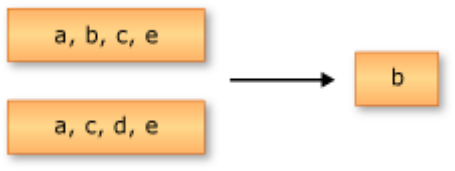
foreach (string word in words.DistinctBy(p => p.Length))
{
    Console.WriteLine(word);
}

// This code produces the following output:
//     the
//     quick
```

```
//     jumped
//     over
```

`Except` and `ExceptBy`

The following example depicts the behavior of Enumerable.Except. The returned sequence contains only the elements from the first input sequence that aren't in the second input sequence.



> **Note**
>
> The following examples in this article use the common data sources for this area. Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves as the department head.
> You can find the example data set in the source repo.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
```

```
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Except(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * quick
 * brown
 * fox
 */
```

The ExceptBy method is an alternative approach to `Except` that takes two sequences of possibly heterogenous types and a `keySelector`. The `keySelector` is the same type as the first collection's type. Consider the following `Teacher` array and teacher IDs to exclude. To find teachers in the first collection that aren't in the second collection, you can project the teacher's ID onto the second collection:

C#

```
int[] teachersToExclude =
[
    901,    // English
    965,    // Mathematics
    932,    // Engineering
    945,    // Economics
    987,    // Physics
```

```
    901     // Chemistry
];

foreach (Teacher teacher in
    teachers.ExceptBy(
        teachersToExclude, teacher => teacher.ID))
{
    Console.WriteLine($"{teacher.First} {teacher.Last}");
}
```

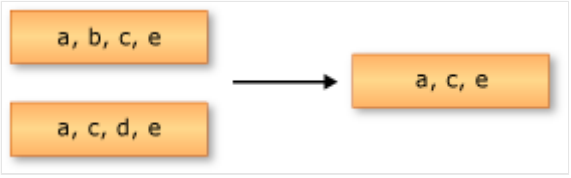
In the preceding C# code:

- The `teachers` array is filtered to only those teachers that aren't in the `teachersToExclude` array.
- The `teachersToExclude` array contains the `ID` value for all department heads.
- The call to `ExceptBy` results in a new set of values that are written to the console.

The new set of values is of type `Teacher`, which is the type of the first collection. Each `teacher` in the `teachers` array that doesn't have a corresponding ID value in the `teachersToExclude` array is written to the console.

`Intersect` and `IntersectBy`

The following example depicts the behavior of Enumerable.Intersect. The returned sequence contains the elements that are common to both of the input sequences.



C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Intersect(words2)
                            select word;

foreach (var str in query)
{
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
```

```
 * the
 */
```

The IntersectBy method is an alternative approach to `Intersect` that takes two sequences of possibly heterogenous types and a `keySelector`. The `keySelector` is used as the comparative discriminator of the second collection's type. Consider the following student and teacher arrays. The query matches items in each sequence by name to find those students who are also teachers:

C#

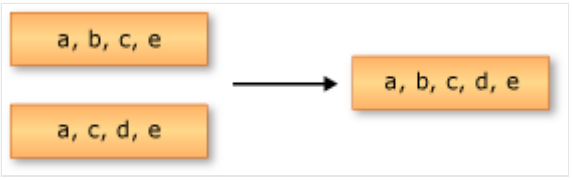
```
foreach (Student person in
    students.IntersectBy(
        teachers.Select(t => (t.First, t.Last)), s => (s.FirstName,
s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

In the preceding C# code:

- The query produces the intersection of the `Teacher` and `Student` by comparing names.
- Only people that are found in both arrays are present in the resulting sequence.
- The resulting `Student` instances are written to the console.

`Union` and `UnionBy`

The following example depicts a union operation on two sequences of strings. The returned sequence contains the unique elements from both input sequences.



C#

```
string[] words1 = ["the", "quick", "brown", "fox"];
string[] words2 = ["jumped", "over", "the", "lazy", "dog"];

IEnumerable<string> query = from word in words1.Union(words2)
                            select word;

foreach (var str in query)
{
```

```
    Console.WriteLine(str);
}

/* This code produces the following output:
 *
 * the
 * quick
 * brown
 * fox
 * jumped
 * over
 * lazy
 * dog
*/
```

The UnionBy method is an alternative approach to `Union` that takes two sequences of the same type and a `keySelector`. The `keySelector` is used as the comparative discriminator of the source type. The following query produces the list of all people that are either students or teachers. Students who are also teachers are added to the union set only once:

C#

```
foreach (var person in
    students.Select(s => (s.FirstName, s.LastName)).UnionBy(
        teachers.Select(t => (FirstName: t.First, LastName: t.Last)), s =>
(s.FirstName, s.LastName)))
{
    Console.WriteLine($"{person.FirstName} {person.LastName}");
}
```

In the preceding C# code:

- The `teachers` and `students` arrays are woven together using their names as the key selector.
- The resulting names are written to the console.

See also

- System.Linq
- How to find the set difference between two lists (LINQ) (C#)

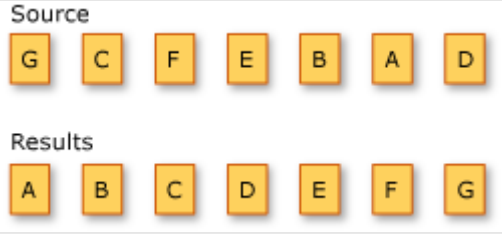
Sorting Data (C#)

08/13/2025

A sorting operation orders the elements of a sequence based on one or more attributes. The first sort criterion performs a primary sort on the elements. By specifying a second sort criterion, you can sort the elements within each primary sort group.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

The following illustration shows the results of an alphabetical sort operation on a sequence of characters:



The standard query operator methods that sort data are listed in the following section.

Methods

Expand table

Method Name	Description	C# Query Expression Syntax	More Information
OrderBy	Sorts values in ascending order.	`orderby`	Enumerable.OrderBy Queryable.OrderBy
OrderByDescending	Sorts values in descending order.	`orderby ... descending`	Enumerable.OrderByDescending Queryable.OrderByDescending

Method Name	Description	C# Query Expression Syntax	More Information
ThenBy	Performs a secondary sort in ascending order.	`orderby …, …`	Enumerable.ThenBy Queryable.ThenBy
ThenByDescending	Performs a secondary sort in descending order.	`orderby …, … descending`	Enumerable.ThenByDescending Queryable.ThenByDescending
Reverse	Reverses the order of the elements in a collection.	Not applicable.	Enumerable.Reverse Queryable.Reverse

> **Note**
>
> The following examples in this article use the common data sources for this area. Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves as the department head. You can find the example data set in the source repo.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
```

```
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

> **Note**
>
> You can refer to the common data sources for this area in the Standard Query Operators Overview article.

Primary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to sort the array of teachers by family name, in ascending order.

C#

```
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

The equivalent query written using method syntax is shown in the following code:

C#

```
IEnumerable<string> query = teachers
    .OrderBy(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

Primary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to sort the teachers by family name, in descending order.

C#

```
IEnumerable<string> query = from teacher in teachers
                            orderby teacher.Last descending
                            select teacher.Last;

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

The equivalent query written using method syntax is shown in the following code:

C#

```
IEnumerable<string> query = teachers
    .OrderByDescending(teacher => teacher.Last)
    .Select(teacher => teacher.Last);

foreach (string str in query)
{
    Console.WriteLine(str);
}
```

Secondary Ascending Sort

The following example demonstrates how to use the `orderby` clause in a LINQ query to perform a primary and secondary sort. The teachers are sorted primarily by city and secondarily by their family name, both in ascending order.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                            orderby teacher.City, teacher.Last
                            select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

The equivalent query written using method syntax is shown in the following code:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenBy(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

Secondary Descending Sort

The next example demonstrates how to use the `orderby descending` clause in a LINQ query to perform a primary sort, in ascending order, and a secondary sort, in descending order. The teachers are sorted primarily by city and secondarily by their family name.

C#

```
IEnumerable<(string, string)> query = from teacher in teachers
                          orderby teacher.City, teacher.Last descending
                          select (teacher.Last, teacher.City);

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

The equivalent query written using method syntax is shown in the following code:

C#

```
IEnumerable<(string, string)> query = teachers
    .OrderBy(teacher => teacher.City)
    .ThenByDescending(teacher => teacher.Last)
    .Select(teacher => (teacher.Last, teacher.City));

foreach ((string last, string city) in query)
{
    Console.WriteLine($"City: {city}, Last Name: {last}");
}
```

See also

- System.Linq
- orderby clause
- How to sort or filter text data by any word or field (LINQ) (C#)

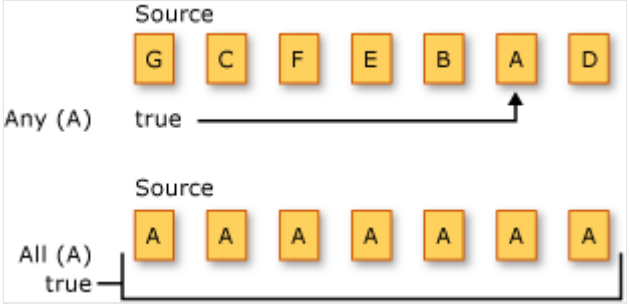
Quantifier operations in LINQ (C#)

08/13/2025

Quantifier operations return a Boolean value that indicates whether some or all of the elements in a sequence satisfy a condition.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

The following illustration depicts two different quantifier operations on two different source sequences. The first operation asks if any of the elements are the character 'A'. The second operation asks if all the elements are the character 'A'. Both methods return `true` in this example.



Expand table

Method Name	Description	C# Query Expression Syntax	More Information
All	Determines whether all the elements in a sequence satisfy a condition.	Not applicable.	Enumerable.All Queryable.All
Any	Determines whether any elements in a sequence satisfy a condition.	Not applicable.	Enumerable.Any Queryable.Any
Contains	Determines whether a sequence contains a specified element.	Not applicable.	Enumerable.Contains Queryable.Contains

All

The following example uses the `All` to find students that scored above 70 on all exams.

> **Note**
>
> You can refer to the common data sources for this area in the Standard Query Operators Overview article.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.All(score => score > 70)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Cesar Garcia: 71, 86, 77, 97
// Nancy Engström: 75, 73, 78, 83
// Ifunanya Ugomma: 84, 82, 96, 80
```

Any

The following example uses the `Any` to find students that scored greater than 95 on any exam.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.Any(score => score > 95)
                            select $"{student.FirstName} {student.LastName}:
{student.Scores.Max()}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Svetlana Omelchenko: 97
// Cesar Garcia: 97
// Debra Garcia: 96
// Ifeanacho Jamuike: 98
// Ifunanya Ugomma: 96
```

```
// Michelle Caruana: 97
// Nwanneka Ifeoma: 98
// Martina Mattsson: 96
// Anastasiya Sazonova: 96
// Jesper Jakobsson: 98
// Max Lindgren: 96
```

Contains

The following example uses the `Contains` to find students that scored exactly 95 on an exam.

C#

```
IEnumerable<string> names = from student in students
                            where student.Scores.Contains(95)
                            select $"{student.FirstName} {student.LastName}:
{string.Join(", ", student.Scores.Select(s => s.ToString()))}";

foreach (string name in names)
{
    Console.WriteLine($"{name}");
}

// This code produces the following output:
//
// Claire O'Donnell: 56, 78, 95, 95
// Donald Urquhart: 92, 90, 95, 57
```

See also

- System.Linq
- Dynamically specify predicate filters at run time
- How to query for sentences that contain a specified set of words (LINQ) (C#)

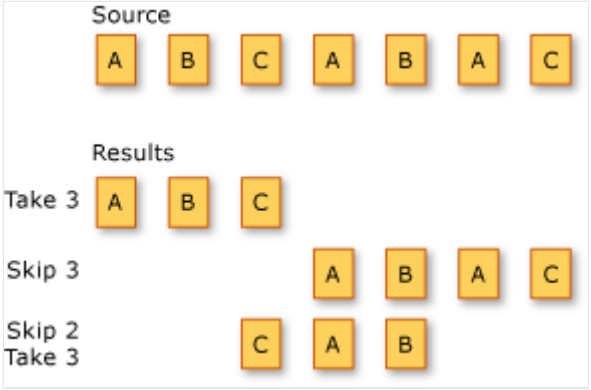
Partitioning data (C#)

08/13/2025

Partitioning in LINQ refers to the operation of dividing an input sequence into two sections, without rearranging the elements, and then returning one of the sections.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

The following illustration shows the results of three different partitioning operations on a sequence of characters. The first operation returns the first three elements in the sequence. The second operation skips the first three elements and returns the remaining elements. The third operation skips the first two elements in the sequence and returns the next three elements.



The standard query operator methods that partition sequences are listed in the following section.

Operators

Expand table

Method names	Description	C# query expression syntax	More information
Skip	Skips elements up to a specified position in a sequence.	Not applicable.	Enumerable.Skip Queryable.Skip

Method names	Description	C# query expression syntax	More information
SkipWhile	Skips elements based on a predicate function until an element doesn't satisfy the condition.	Not applicable.	Enumerable.SkipWhile Queryable.SkipWhile
Take	Takes elements up to a specified position in a sequence.	Not applicable.	Enumerable.Take Queryable.Take
TakeWhile	Takes elements based on a predicate function until an element doesn't satisfy the condition.	Not applicable.	Enumerable.TakeWhile Queryable.TakeWhile
Chunk	Splits the elements of a sequence into chunks of a specified maximum size.	Not applicable.	Enumerable.Chunk Queryable.Chunk

All the following examples use Enumerable.Range(Int32, Int32) to generate a sequence of numbers from 0 through 7.

> **Note**
>
> You can refer to the common data sources for this area in the Standard Query Operators Overview article.

You use the `Take` method to take only the first elements in a sequence:

C#

```
foreach (int number in Enumerable.Range(0, 8).Take(3))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
```

You use the `Skip` method to skip the first elements in a sequence, and use the remaining elements:

C#

```
foreach (int number in Enumerable.Range(0, 8).Skip(3))
{
    Console.WriteLine(number);
```

```
}
// This code produces the following output:
// 3
// 4
// 5
// 6
// 7
```

The `TakeWhile` and `SkipWhile` methods also take and skip elements in a sequence. However, instead of a set number of elements, these methods skip or take elements based on a condition. `TakeWhile` takes the elements of a sequence until an element doesn't match the condition.

C#

```
foreach (int number in Enumerable.Range(0, 8).TakeWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 0
// 1
// 2
// 3
// 4
```

`SkipWhile` skips the first elements, as long as the condition is true. The first element not matching the condition, and all subsequent elements, are returned.

C#

```
foreach (int number in Enumerable.Range(0, 8).SkipWhile(n => n < 5))
{
    Console.WriteLine(number);
}
// This code produces the following output:
// 5
// 6
// 7
```

The `Chunk` operator is used to split elements of a sequence based on a given `size`.

C#

```
int chunkNumber = 1;
foreach (int[] chunk in Enumerable.Range(0, 8).Chunk(3))
{
    Console.WriteLine($"Chunk {chunkNumber++}:");
```

```
    foreach (int item in chunk)
    {
        Console.WriteLine($"    {item}");
    }

    Console.WriteLine();
}
// This code produces the following output:
// Chunk 1:
//    0
//    1
//    2
//
//Chunk 2:
//    3
//    4
//    5
//
//Chunk 3:
//    6
//    7
```

The preceding C# code:

- Relies on Enumerable.Range(Int32, Int32) to generate a sequence of numbers.
- Applies the `Chunk` operator, splitting the sequence into chunks with a max size of three.

See also

- System.Linq

Converting Data Types (C#)

Article • 10/08/2024

Conversion methods change the type of input objects.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

Conversion operations in LINQ queries are useful in various applications. Following are some examples:

- The Enumerable.AsEnumerable method can be used to hide a type's custom implementation of a standard query operator.
- The Enumerable.OfType method can be used to enable non-parameterized collections for LINQ querying.
- The Enumerable.ToArray, Enumerable.ToDictionary, Enumerable.ToList, and Enumerable.ToLookup methods can be used to force immediate query execution instead of deferring it until the query is enumerated.

Methods

The following table lists the standard query operator methods that perform data-type conversions.

The conversion methods in this table whose names start with "As" change the static type of the source collection but don't enumerate it. The methods whose names start with "To" enumerate the source collection and put the items into the corresponding collection type.

Expand table

Method Name	Description	C# Query Expression Syntax	More Information
AsEnumerable	Returns the input typed as IEnumerable<T>.	Not applicable.	Enumerable.AsEnumerable
AsQueryable	Converts a (generic) IEnumerable to a (generic) IQueryable.	Not applicable.	Queryable.AsQueryable
Cast	Casts the elements of a collection to a specified type.	Use an explicitly typed range variable. For example: `from string str in words`	Enumerable.Cast Queryable.Cast
OfType	Filters values, depending on their ability to be cast to a specified type.	Not applicable.	Enumerable.OfType Queryable.OfType
ToArray	Converts a collection to an array. This method forces query execution.	Not applicable.	Enumerable.ToArray
ToDictionary	Puts elements into a Dictionary<TKey,TValue> based on a key selector function. This method forces query execution.	Not applicable.	Enumerable.ToDictionary
ToList	Converts a collection to a List<T>. This method forces query execution.	Not applicable.	Enumerable.ToList
ToLookup	Puts elements into a Lookup<TKey,TElement> (a one-to-many dictionary) based on a key selector function. This method forces query execution.	Not applicable.	Enumerable.ToLookup

> **Note**
>
> The following examples in this article use the common data sources for this area. Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves

as the department head.
You can find the example data set in the source repo.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Query Expression Syntax Example

The following code example uses an explicitly typed range variable to cast a type to a subtype before accessing a member that is available only on the subtype.

C#

```
IEnumerable people = students;

var query = from Student student in students
            where student.Year == GradeLevel.ThirdYear
            select student;

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

The equivalent query can be expressed using method syntax as shown in the following example:

C#

```
IEnumerable people = students;

var query = people
    .Cast<Student>()
    .Where(student => student.Year == GradeLevel.ThirdYear);

foreach (Student student in query)
{
    Console.WriteLine(student.FirstName);
}
```

See also

- System.Linq
- from clause

Join operations in LINQ

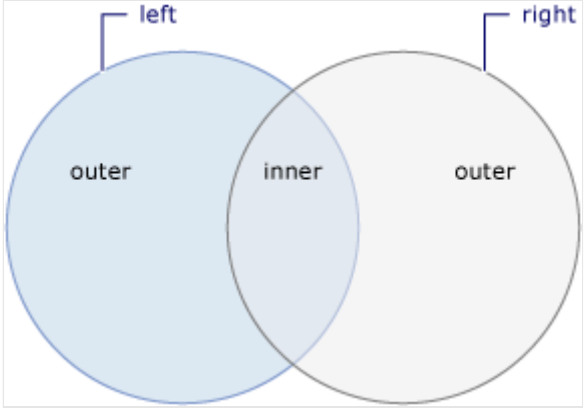
A *join* associates objects in one data source with objects that share a common attribute in another data source.

> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

Joining is an important operation in queries that target data sources whose relationships to each other you can't follow directly. In object-oriented programming, joining could mean a correlation between objects that isn't modeled, such as the backwards direction of a one-way relationship. An example of a one-way relationship is a `Student` class that has a property of type `Department` that represents the major, but the `Department` class doesn't have a property that is a collection of `Student` objects. If you have a list of `Department` objects and you want to find all the students in each department, you could use a join operation to find them.

The LINQ framework provides join methods: Join and GroupJoin. These methods perform equijoins, or joins that match two data sources based on equality of their keys. For comparison, Transact-SQL supports join operators other than `equals`, such as the `less than` operator. In relational database terms, Join implements an inner join, a type of join in which only those objects that have a match in the other data set are returned. The GroupJoin method has no direct equivalent in relational database terms, but it implements a superset of inner joins and left outer joins. A left outer join is a join that returns each element of the first (left) data source, even if it has no correlated elements in the other data source.

The following illustration shows a conceptual view of two sets and the elements within those sets that are included in either an inner join or a left outer join.



Methods

Expand table

Method Name	Description	C# Query Expression Syntax	More Information
Join	Joins two sequences based on key selector functions and extracts pairs of values.	`join … in … on … equals …`	Enumerable.Join Queryable.Join
GroupJoin	Joins two sequences based on key selector functions and groups the resulting matches for each element.	`join … in … on … equals … into …`	Enumerable.GroupJoin Queryable.GroupJoin
LeftJoin	Correlates the elements of two sequences based on matching keys.	N/A	Enumerable.LeftJoin Queryable.LeftJoin
RightJoin	Correlates the elements of two sequences based on matching keys.	N/A	Enumerable.RightJoin Queryable.RightJoin

> **Note**
>
> The following examples in this article use the common data sources for this area. Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves as the department head. You can find the example data set in the source repo.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }
```

```
    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

> **Note**
>
> You can refer to the common data sources for this area in the Standard Query Operators Overview article.

The following example uses the `join … in … on … equals …` clause to join two sequences based on a specific value:

C#

```
var query = from student in students
            join department in departments on student.DepartmentID equals
department.ID
            select new { Name = $"{student.FirstName} {student.LastName}",
DepartmentName = department.Name };

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

You can express the preceding query by using method syntax, as shown in the following code:

C#

```
var query = students.Join(departments,
    student => student.DepartmentID, department => department.ID,
```

```
    (student, department) => new { Name = $"{student.FirstName} {student.LastName}",
DepartmentName = department.Name });

foreach (var item in query)
{
    Console.WriteLine($"{item.Name} - {item.DepartmentName}");
}
```

The following example uses the `join … in … on … equals … into …` clause to join two sequences based on a specific value and group the resulting matches for each element:

C#

```
IEnumerable<IEnumerable<Student>> studentGroups = from department in departments
                    join student in students on department.ID equals
student.DepartmentID into studentGroup
                    select studentGroup;

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
    }
}
```

You can express the preceding query by using method syntax, as shown in the following example:

C#

```
// Join department and student based on DepartmentId and grouping result
IEnumerable<IEnumerable<Student>> studentGroups = departments.GroupJoin(students,
    department => department.ID, student => student.DepartmentID,
    (department, studentGroup) => studentGroup);

foreach (IEnumerable<Student> studentGroup in studentGroups)
{
    Console.WriteLine("Group");
    foreach (Student student in studentGroup)
    {
        Console.WriteLine($"  - {student.FirstName}, {student.LastName}");
    }
}
```

Perform inner joins

In relational database terms, an *inner join* produces a result set in which each element of the first collection appears one time for every matching element in the second collection. If an element in the first collection has no matching elements, it doesn't appear in the result set. The Join method, which the `join` clause in C# calls, implements an inner join. The following examples show you how to perform four variations of an inner join:

- A simple inner join that correlates elements from two data sources based on a simple key.
- An inner join that correlates elements from two data sources based on a *composite* key. A composite key, which is a key that consists of more than one value, enables you to correlate elements based on more than one property.
- A *multiple join* in which you append successive join operations to each other.
- An inner join that uses a group join.

Single key join

The following example matches `Teacher` objects with `Department` objects whose `TeacherId` matches that `Teacher`. The `select` clause in C# defines how the resulting objects look. In the following example, the resulting objects are anonymous types that consist of the department name and the name of the teacher that leads the department.

C#

```
var query = from department in departments
            join teacher in teachers on department.TeacherID equals teacher.ID
            select new
            {
                DepartmentName = department.Name,
                TeacherName = $"{teacher.First} {teacher.Last}"
            };

foreach (var departmentAndTeacher in query)
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

You achieve the same results by using the Join method syntax:

C#

```
var query = teachers
    .Join(departments, teacher => teacher.ID, department => department.TeacherID,
        (teacher, department) =>
        new { DepartmentName = department.Name, TeacherName = $"{teacher.First}
{teacher.Last}" });

foreach (var departmentAndTeacher in query)
```

```
{
    Console.WriteLine($"{departmentAndTeacher.DepartmentName} is managed by
{departmentAndTeacher.TeacherName}");
}
```

Teachers who aren't department heads don't appear in the final results.

Composite key join

Instead of correlating elements based on just one property, use a composite key to compare elements based on multiple properties. Specify the key selector function for each collection to return an anonymous type that consists of the properties you want to compare. If you label the properties, they must have the same label in each key's anonymous type. The properties must also appear in the same order.

The following example uses a list of `Teacher` objects and a list of `Student` objects to determine which teachers are also students. Both of these types have properties that represent the first and family name of each person. The functions that create the join keys from each list's elements return an anonymous type that consists of the properties. The join operation compares these composite keys for equality and returns pairs of objects from each list where both the first name and the family name match.

C#

```
// Join the two data sources based on a composite key consisting of first and last
name,
// to determine which employees are also students.
IEnumerable<string> query =
    from teacher in teachers
    join student in students on new
    {
        FirstName = teacher.First,
        LastName = teacher.Last
    } equals new
    {
        student.FirstName,
        student.LastName
    }
    select teacher.First + " " + teacher.Last;

string result = "The following people are both teachers and students:\r\n";
foreach (string name in query)
{
    result += $"{name}\r\n";
}
Console.Write(result);
```

You can use the Join method, as shown in the following example:

C#

```
IEnumerable<string> query = teachers
    .Join(students,
        teacher => new { FirstName = teacher.First, LastName = teacher.Last },
        student => new { student.FirstName, student.LastName },
        (teacher, student) => $"{teacher.First} {teacher.Last}"
 );

Console.WriteLine("The following people are both teachers and students:");
foreach (string name in query)
{
    Console.WriteLine(name);
}
```

Multiple join

You can append any number of join operations to perform a multiple join. Each `join` clause in C# correlates a specified data source with the results of the previous join.

The first `join` clause matches students and departments based on a `Student` object's `DepartmentID` matching a `Department` object's `ID`. It returns a sequence of anonymous types that contain the `Student` object and `Department` object.

The second `join` clause correlates the anonymous types returned by the first join with `Teacher` objects based on that teacher's ID matching the department head ID. It returns a sequence of anonymous types that contain the student's name, the department name, and the department leader's name. Because this operation is an inner join, the query returns only those objects from the first data source that have a match in the second data source.

C#

```
// The first join matches Department.ID and Student.DepartmentID from the list of
students and
// departments, based on a common ID. The second join matches teachers who lead
departments
// with the students studying in that department.
var query = from student in students
    join department in departments on student.DepartmentID equals department.ID
    join teacher in teachers on department.TeacherID equals teacher.ID
    select new {
        StudentName = $"{student.FirstName} {student.LastName}",
        DepartmentName = department.Name,
        TeacherName = $"{teacher.First} {teacher.Last}"
    };

foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the department
```

```
run by "{obj.TeacherName}".""");
}
```

The equivalent query that uses multiple Join methods uses the same approach with the anonymous type:

C#

```
var query = students
    .Join(departments, student => student.DepartmentID, department => department.ID,
        (student, department) => new { student, department })
    .Join(teachers, commonDepartment => commonDepartment.department.TeacherID,
teacher => teacher.ID,
        (commonDepartment, teacher) => new
        {
            StudentName = $"{commonDepartment.student.FirstName}
{commonDepartment.student.LastName}",
            DepartmentName = commonDepartment.department.Name,
            TeacherName = $"{teacher.First} {teacher.Last}"
        });

foreach (var obj in query)
{
    Console.WriteLine($"""The student "{obj.StudentName}" studies in the department
run by "{obj.TeacherName}".""");
}
```

Inner join by using grouped join

The following example shows how to implement an inner join by using a group join. The list of `Department` objects is group-joined to the list of `Student` objects based on the `Department.ID` matching the `Student.DepartmentID` property. The group join creates a collection of intermediate groups, where each group consists of a `Department` object and a sequence of matching `Student` objects. The second `from` clause combines (or flattens) this sequence of sequences into one longer sequence. The `select` clause specifies the type of elements in the final sequence. That type is an anonymous type that consists of the student's name and the matching department name.

C#

```
var query1 =
    from department in departments
    join student in students on department.ID equals student.DepartmentID into gj
    from subStudent in gj
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
```

```
    };
Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in query1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

You can achieve the same results by using the GroupJoin method, as shown in the following example:

C#

```
var queryMethod1 = departments
    .GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, gj) => new { department, gj })
    .SelectMany(departmentAndStudent => departmentAndStudent.gj,
        (departmentAndStudent, subStudent) => new
        {
            DepartmentName = departmentAndStudent.department.Name,
            StudentName = $"{subStudent.FirstName} {subStudent.LastName}"
        });

Console.WriteLine("Inner join using GroupJoin():");
foreach (var v in queryMethod1)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

The result is equivalent to the result set obtained by using the `join` clause without the `into` clause to perform an inner join. The following code demonstrates this equivalent query:

C#

```
var query2 = from department in departments
    join student in students on department.ID equals student.DepartmentID
    select new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    };

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in query2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

To avoid chaining, use the single Join method as presented here:

C#

```
var queryMethod2 = departments.Join(students, departments => departments.ID, student
=> student.DepartmentID,
    (department, student) => new
    {
        DepartmentName = department.Name,
        StudentName = $"{student.FirstName} {student.LastName}"
    });

Console.WriteLine("The equivalent operation using Join():");
foreach (var v in queryMethod2)
{
    Console.WriteLine($"{v.DepartmentName} - {v.StudentName}");
}
```

Perform grouped joins

The group join is useful for producing hierarchical data structures. It pairs each element from the first collection with a set of correlated elements from the second collection.

> ⓘ **Note**
>
> Each element of the first collection appears in the result set of a group join regardless of whether correlated elements are found in the second collection. If no correlated elements are found, the sequence of correlated elements for that element is empty. The result selector therefore has access to every element of the first collection. This behavior differs from the result selector in a non-group join, which can't access elements from the first collection that have no match in the second collection.

> ⚠ **Warning**
>
> Enumerable.GroupJoin has no direct equivalent in traditional relational database terms. However, this method does implement a superset of inner joins and left outer joins. Both of these operations can be written in terms of a grouped join. For more information, see Entity Framework Core, GroupJoin.

The first example in this article shows how to perform a group join. The second example shows how to use a group join to create XML elements.

Group join

The following example performs a group join of objects of type `Department` and `Student` based on the `Department.ID` matching the `Student.DepartmentID` property. Unlike a non-group join, which produces a pair of elements for each match, the group join produces only one resulting object for each element of the first collection. In this example, the first collection is a `Department` object. The corresponding elements from the second collection, which in this example are `Student` objects, are grouped into a collection. Finally, the result selector function creates an anonymous type for each match that consists of `Department.Name` and a collection of `Student` objects.

C#

```
var query = from department in departments
    join student in students on department.ID equals student.DepartmentID into
studentGroup
    select new
    {
        DepartmentName = department.Name,
        Students = studentGroup
    };

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");

    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
    }
}
```

In the preceding example, the `query` variable contains the query that creates a list where each element is an anonymous type that contains the department's name and a collection of students that study in that department.

The equivalent query using method syntax is shown in the following code:

C#

```
var query = departments.GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
    (department, Students) => new { DepartmentName = department.Name, Students });

foreach (var v in query)
{
    // Output the department's name.
    Console.WriteLine($"{v.DepartmentName}:");
```

```
    // Output each of the students in that department.
    foreach (Student? student in v.Students)
    {
        Console.WriteLine($"  {student.FirstName} {student.LastName}");
    }
}
```

Group join to create XML

Group joins are ideal for creating XML by using LINQ to XML. The following example is similar to the previous example except that instead of creating anonymous types, the result selector function creates XML elements that represent the joined objects.

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    from department in departments
    join student in students on department.ID equals student.DepartmentID into
studentGroup
    select new XElement("Department",
        new XAttribute("Name", department.Name),
        from student in studentGroup
        select new XElement("Student",
            new XAttribute("FirstName", student.FirstName),
            new XAttribute("LastName", student.LastName)
        )
    )
);

Console.WriteLine(departmentsAndStudents);
```

The equivalent query using method syntax is shown in the following code:

C#

```
XElement departmentsAndStudents = new("DepartmentEnrollment",
    departments.GroupJoin(students, department => department.ID, student =>
student.DepartmentID,
        (department, Students) => new XElement("Department",
            new XAttribute("Name", department.Name),
            from student in Students
            select new XElement("Student",
                new XAttribute("FirstName", student.FirstName),
                new XAttribute("LastName", student.LastName)
            )
        )
    )
);

Console.WriteLine(departmentsAndStudents);
```

Perform outer joins

.NET 10 includes LeftJoin and RightJoin methods in the System.Linq.Enumerable and System.Linq.Queryable classes. These methods perform an *outer left equijoin*, and an *outer right equijoin*, respectively. An outer left equijoin is a join where every member of the first sequence is included in the output sequence, even if the second sequence doesn't include a match. An outer right equijoin is a join where every member of the second sequence is included in the output sequence, even if the first sequence doesn't include a match.

Emulate a left outer join

Before .NET 10, use LINQ to perform a left outer join by calling the DefaultIfEmpty method on the results of a group join.

The following example demonstrates how to use the DefaultIfEmpty method on the results of a group join to perform a left outer join.

The first step in producing a left outer join of two collections is to perform an inner join by using a group join. (See Perform inner joins for an explanation of this process.) In this example, the list of `Department` objects is inner-joined to the list of `Student` objects based on a `Department` object's ID that matches the student's `DepartmentID`.

The second step is to include each element of the first (left) collection in the result set even if that element has no matches in the right collection. You accomplish this step by calling DefaultIfEmpty on each sequence of matching elements from the group join. In this example, you call DefaultIfEmpty on each sequence of matching `Student` objects. The method returns a collection that contains a single, default value if the sequence of matching `Student` objects is empty for any `Department` object, ensuring that each `Department` object is represented in the result collection.

> **Note**
>
> The default value for a reference type is `null`; therefore, the example checks for a null reference before accessing each element of each `Student` collection.

C#

```
var query =
    from student in students
    join department in departments on student.DepartmentID equals department.ID into
gj
    from subgroup in gj.DefaultIfEmpty()
```

```
    select new
    {
        student.FirstName,
        student.LastName,
        Department = subgroup?.Name ?? string.Empty
    };

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}: {v.Department}");
}
```

The equivalent query using method syntax is shown in the following code:

C#

```
var query = students
    .GroupJoin(
        departments,
        student => student.DepartmentID,
        department => department.ID,
        (student, departmentList) => new { student, subgroup = departmentList })
    .SelectMany(
        joinedSet => joinedSet.subgroup.DefaultIfEmpty(),
        (student, department) => new
        {
            student.student.FirstName,
            student.student.LastName,
            Department = department?.Name ?? string.Empty
        });

foreach (var v in query)
{
    Console.WriteLine($"{v.FirstName:-15} {v.LastName:-15}: {v.Department}");
}
```

See also

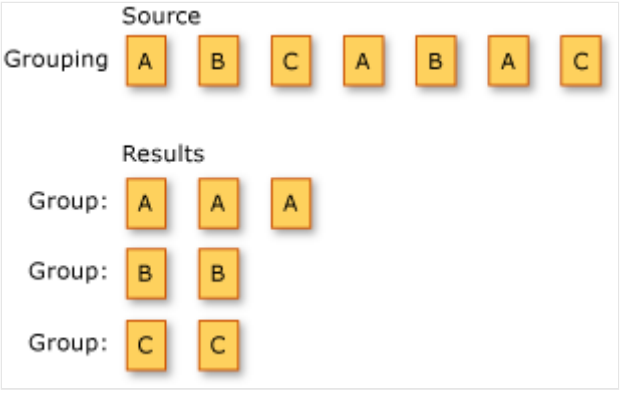
- Join
- GroupJoin
- Anonymous types
- Formulate Joins and Cross-Product Queries
- join clause
- group clause
- How to join content from dissimilar files (LINQ) (C#)
- How to populate object collections from multiple sources (LINQ) (C#)

Last updated on 12/16/2025

Grouping Data (C#)

Article • 05/31/2024

Grouping refers to the operation of putting data into groups so that the elements in each group share a common attribute. The following illustration shows the results of grouping a sequence of characters. The key for each group is the character.



> **Important**
>
> These samples use an System.Collections.Generic.IEnumerable<T> data source. Data sources based on System.Linq.IQueryProvider use System.Linq.IQueryable<T> data sources and expression trees. Expression trees have limitations on the allowed C# syntax. Furthermore, each `IQueryProvider` data source, such as EF Core may impose more restrictions. Check the documentation for your data source.

The standard query operator methods that group data elements are listed in the following table.

Expand table

Method Name	Description	C# Query Expression Syntax	More Information

GroupBy	Groups elements that share a common attribute. An IGrouping<TKey,TElement> object represents each group.	`group … by`
		-or-
		`group … by … into …`
			Enumerable.GroupBy
			Queryable.GroupBy

Method Name	Description	C# Query Expression Syntax	More Information
ToLookup	Inserts elements into a Lookup<TKey,TElement> (a one-to-many dictionary) based on a key selector function.	Not applicable.	Enumerable.ToLookup

The following code example uses the `group by` clause to group integers in a list according to whether they're even or odd.

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = from number in numbers
                                         group number by number % 2;

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

The equivalent query using method syntax is shown in the following code:

C#

```
List<int> numbers = [35, 44, 200, 84, 3987, 4, 199, 329, 446, 208];

IEnumerable<IGrouping<int, int>> query = numbers
    .GroupBy(number => number % 2);

foreach (var group in query)
{
    Console.WriteLine(group.Key == 0 ? "\nEven numbers:" : "\nOdd
numbers:");
    foreach (int i in group)
    {
        Console.WriteLine(i);
    }
}
```

> ⓘ Note

The following examples in this article use the common data sources for this area. Each `Student` has a grade level, a primary department, and a series of scores. A `Teacher` also has a `City` property that identifies the campus where the teacher holds classes. A `Department` has a name, and a reference to a `Teacher` who serves as the department head.
You can find the example data set in the source repo.

C#

```
public enum GradeLevel
{
    FirstYear = 1,
    SecondYear,
    ThirdYear,
    FourthYear
};

public class Student
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
    public required int ID { get; init; }

    public required GradeLevel Year { get; init; }
    public required List<int> Scores { get; init; }

    public required int DepartmentID { get; init; }
}

public class Teacher
{
    public required string First { get; init; }
    public required string Last { get; init; }
    public required int ID { get; init; }
    public required string City { get; init; }
}

public class Department
{
    public required string Name { get; init; }
    public int ID { get; init; }

    public required int TeacherID { get; init; }
}
```

Group query results

Grouping is one of the most powerful capabilities of LINQ. The following examples show how to group data in various ways:

- By a single property.
- By the first letter of a string property.
- By a computed numeric range.
- By Boolean predicate or other expression.
- By a compound key.

In addition, the last two queries project their results into a new anonymous type that contains only the student's first and family name. For more information, see the group clause.

Group by single property example

The following example shows how to group source elements by using a single property of the element as the group key. The key is an `enum`, the student's year in school. The grouping operation uses the default equality comparer for the type.

C#

```
var groupByYearQuery =
    from student in students
    group student by student.Year into newGroup
    orderby newGroup.Key
    select newGroup;

foreach (var yearGroup in groupByYearQuery)
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

The equivalent code using method syntax is shown in the following example:

C#

```
// Variable groupByLastNamesQuery is an IEnumerable<IGrouping<string,
// DataClass.Student>>.
var groupByYearQuery = students
    .GroupBy(student => student.Year)
    .OrderBy(newGroup => newGroup.Key);

foreach (var yearGroup in groupByYearQuery)
```

```
{
    Console.WriteLine($"Key: {yearGroup.Key}");
    foreach (var student in yearGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Group by value example

The following example shows how to group source elements by using something other than a property of the object for the group key. In this example, the key is the first letter of the student's family name.

C#

```
var groupByFirstLetterQuery =
    from student in students
    let firstLetter = student.LastName[0]
    group student by firstLetter;

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Nested foreach is required to access group items.

The equivalent code using method syntax is shown in the following example:

C#

```
var groupByFirstLetterQuery = students
    .GroupBy(student => student.LastName[0]);

foreach (var studentGroup in groupByFirstLetterQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.LastName}, {student.FirstName}");
    }
}
```

Group by a range example

The following example shows how to group source elements by using a numeric range as a group key. The query then projects the results into an anonymous type that contains only the first and family name and the percentile range to which the student belongs. An anonymous type is used because it isn't necessary to use the complete `Student` object to display the results. `GetPercentile` is a helper function that calculates a percentile based on the student's average score. The method returns an integer between 0 and 10.

C#

```
static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}

var groupByPercentileQuery =
    from student in students
    let percentile = GetPercentile(student)
    group new
    {
        student.FirstName,
        student.LastName
    } by percentile into percentGroup
    orderby percentGroup.Key
    select percentGroup;

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

Nested foreach required to iterate over groups and group items. The equivalent code using method syntax is shown in the following example:

C#

```
static int GetPercentile(Student s)
{
    double avg = s.Scores.Average();
    return avg > 0 ? (int)avg / 10 : 0;
}
```

```
var groupByPercentileQuery = students
    .Select(student => new { student, percentile = GetPercentile(student) })
    .GroupBy(student => student.percentile)
    .Select(percentGroup => new
    {
        percentGroup.Key,
        Students = percentGroup.Select(s => new { s.student.FirstName,
s.student.LastName })
    })
    .OrderBy(percentGroup => percentGroup.Key);

foreach (var studentGroup in groupByPercentileQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key * 10}");
    foreach (var item in studentGroup.Students)
    {
        Console.WriteLine($"\t{item.LastName}, {item.FirstName}");
    }
}
```

Group by comparison example

The following example shows how to group source elements by using a Boolean comparison expression. In this example, the Boolean expression tests whether a student's average exam score is greater than 75. As in previous examples, the results are projected into an anonymous type because the complete source element isn't needed. The properties in the anonymous type become properties on the `Key` member.

C#

```
var groupByHighAverageQuery =
    from student in students
    group new
    {
        student.FirstName,
        student.LastName
    } by student.Scores.Average() > 75 into studentGroup
    select studentGroup;

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
```

The equivalent query using method syntax is shown in the following code:

C#

```
var groupByHighAverageQuery = students
    .GroupBy(student => student.Scores.Average() > 75)
    .Select(group => new
    {
        group.Key,
        Students = group.AsEnumerable().Select(s => new { s.FirstName,
s.LastName })
    });

foreach (var studentGroup in groupByHighAverageQuery)
{
    Console.WriteLine($"Key: {studentGroup.Key}");
    foreach (var student in studentGroup.Students)
    {
        Console.WriteLine($"\t{student.FirstName} {student.LastName}");
    }
}
```

Group by anonymous type

The following example shows how to use an anonymous type to encapsulate a key that contains multiple values. In this example, the first key value is the first letter of the student's family name. The second key value is a Boolean that specifies whether the student scored over 85 on the first exam. You can order the groups by any property in the key.

C#

```
var groupByCompoundKey =
    from student in students
    group student by new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    } into studentGroup
    orderby studentGroup.Key.FirstLetterOfLastName
    select studentGroup;

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
```

```
    }
}
```

The equivalent query using method syntax is shown in the following code:

C#

```
var groupByCompoundKey = students
    .GroupBy(student => new
    {
        FirstLetterOfLastName = student.LastName[0],
        IsScoreOver85 = student.Scores[0] > 85
    })
    .OrderBy(studentGroup => studentGroup.Key.FirstLetterOfLastName);

foreach (var scoreGroup in groupByCompoundKey)
{
    var s = scoreGroup.Key.IsScoreOver85 ? "more than 85" : "less than 85";
    Console.WriteLine($"Name starts with
{scoreGroup.Key.FirstLetterOfLastName} who scored {s}");
    foreach (var item in scoreGroup)
    {
        Console.WriteLine($"\t{item.FirstName} {item.LastName}");
    }
}
```

Create a nested group

The following example shows how to create nested groups in a LINQ query expression. Each group that is created according to student year or grade level is then further subdivided into groups based on the individuals' names.

C#

```
var nestedGroupsQuery =
    from student in students
    group student by student.Year into newGroup1
    from newGroup2 in
    from student in newGroup1
    group student by student.LastName
    group newGroup2 by newGroup1.Key;

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
```

```
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}
```

Three nested `foreach` loops are required to iterate over the inner elements of a nested group.
(Hover the mouse cursor over the iteration variables, `outerGroup`, `innerGroup`, and `innerGroupElement` to see their actual type.)

The equivalent query using method syntax is shown in the following code:

C#

```
var nestedGroupsQuery =
    students
    .GroupBy(student => student.Year)
    .Select(newGroup1 => new
    {
        newGroup1.Key,
        NestedGroup = newGroup1
            .GroupBy(student => student.LastName)
    });

foreach (var outerGroup in nestedGroupsQuery)
{
    Console.WriteLine($"DataClass.Student Level = {outerGroup.Key}");
    foreach (var innerGroup in outerGroup.NestedGroup)
    {
        Console.WriteLine($"\tNames that begin with: {innerGroup.Key}");
        foreach (var innerGroupElement in innerGroup)
        {
            Console.WriteLine($"\t\t{innerGroupElement.LastName}
{innerGroupElement.FirstName}");
        }
    }
}
```

Perform a subquery on a grouping operation

This article shows two different ways to create a query that orders the source data into groups, and then performs a subquery over each group individually. The basic technique in each example is to group the source elements by using a *continuation* named `newGroup`, and then generating a new subquery against `newGroup`. This subquery is run

against each new group created by the outer query. In this particular example the final output isn't a group, but a flat sequence of anonymous types.

For more information about how to group, see group clause. For more information about continuations, see into. The following example uses an in-memory data structure as the data source, but the same principles apply for any kind of LINQ data source.

C#

```
var queryGroupMax =
    from student in students
    group student by student.Year into studentGroup
    select new
    {
        Level = studentGroup.Key,
        HighestScore = (
            from student2 in studentGroup
            select student2.Scores.Average()
        ).Max()
    };

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

The query in the preceding snippet can also be written using method syntax. The following code snippet has a semantically equivalent query written using method syntax.

C#

```
var queryGroupMax =
    students
        .GroupBy(student => student.Year)
        .Select(studentGroup => new
        {
            Level = studentGroup.Key,
            HighestScore = studentGroup.Max(student2 =>
student2.Scores.Average())
        });

var count = queryGroupMax.Count();
Console.WriteLine($"Number of groups = {count}");

foreach (var item in queryGroupMax)
{
```

```
    Console.WriteLine($"  {item.Level} Highest Score={item.HighestScore}");
}
```

See also

- System.Linq
- GroupBy
- IGrouping<TKey,TElement>
- group clause
- How to split a file into many files by using groups (LINQ) (C#)

How to: Use LINQ to query files and directories

Article • 04/25/2024

Many file system operations are essentially queries and are therefore well suited to the LINQ approach. These queries are nondestructive. They don't change the contents of the original files or folders. Queries shouldn't cause any side-effects. In general, any code (including queries that perform create / update / delete operations) that modifies source data should be kept separate from the code that just queries the data.

There's some complexity involved in creating a data source that accurately represents the contents of the file system and handles exceptions gracefully. The examples in this section create a snapshot collection of FileInfo objects that represents all the files under a specified root folder and all its subfolders. The actual state of each FileInfo might change in the time between when you begin and end executing a query. For example, you can create a list of FileInfo objects to use as a data source. If you try to access the `Length` property in a query, the FileInfo object tries to access the file system to update the value of `Length`. If the file no longer exists, you get a FileNotFoundException in your query, even though you aren't querying the file system directly.

How to query for files with a specified attribute or name

This example shows how to find all files that have a specified file name extension (for example ".txt") in a specified directory tree. It also shows how to return either the newest or oldest file in the tree based on the creation time. You might need to modify the first line of many of the samples whether you're running this code on either Windows, Mac, or a Linux system.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);
var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var fileQuery = from file in fileList
                where file.Extension == ".txt"
                orderby file.Name
                select file;
```

```
// Uncomment this block to see the full query
// foreach (FileInfo fi in fileQuery)
// {
//    Console.WriteLine(fi.FullName);
// }

var newestFile = (from file in fileQuery
                  orderby file.CreationTime
                  select new { file.FullName, file.CreationTime })
                  .Last();

Console.WriteLine($"\r\nThe newest .txt file is {newestFile.FullName}.
Creation time: {newestFile.CreationTime}");
```

How to group files by extension

This example shows how LINQ can be used to perform advanced grouping and sorting operations on lists of files or folders. It also shows how to page output in the console window by using the Skip and Take methods.

The following query shows how to group the contents of a specified directory tree by the file name extension.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

int trimLength = startFolder.Length;

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

var queryGroupByExt = from file in fileList
                      group file by file.Extension.ToLower() into fileGroup
                      orderby fileGroup.Count(), fileGroup.Key
                      select fileGroup;

// Iterate through the outer collection of groups.
foreach (var filegroup in queryGroupByExt.Take(5))
{
    Console.WriteLine($"Extension: {filegroup.Key}");
    var resultPage = filegroup.Take(20);

    //Execute the resultPage query
    foreach (var f in resultPage)
    {
```

```
            Console.WriteLine($"\t{f.FullName.Substring(trimLength)}");
        }
        Console.WriteLine();
    }
```

The output from this program can be long, depending on the details of the local file system and what the `startFolder` is set to. To enable viewing of all results, this example shows how to page through results. A nested `foreach` loop is required because each group is enumerated separately.

How to query for the total number of bytes in a set of folders

This example shows how to retrieve the total number of bytes used by all the files in a specified folder and all its subfolders. The Sum method adds the values of all the items selected in the `select` clause. You can modify this query to retrieve the biggest or smallest file in the specified directory tree by calling the Min or Max method instead of Sum.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

var fileList = Directory.GetFiles(startFolder, "*.*",
SearchOption.AllDirectories);

var fileQuery = from file in fileList
                let fileLen = new FileInfo(file).Length
                where fileLen > 0
                select fileLen;

// Cache the results to avoid multiple trips to the file system.
long[] fileLengths = fileQuery.ToArray();

// Return the size of the largest file
long largestFile = fileLengths.Max();

// Return the total number of bytes in all the files under the specified
folder.
long totalBytes = fileLengths.Sum();

Console.WriteLine($"There are {totalBytes} bytes in {fileList.Count()} files
under {startFolder}");
Console.WriteLine($"The largest file is {largestFile} bytes.");
```

This example extends the previous example to do the following:

- How to retrieve the size in bytes of the largest file.
- How to retrieve the size in bytes of the smallest file.
- How to retrieve the FileInfo object largest or smallest file from one or more folders under a specified root folder.
- How to retrieve a sequence such as the 10 largest files.
- How to order files into groups based on their file size in bytes, ignoring files that are less than a specified size.

The following example contains five separate queries that show how to query and group files, depending on their file size in bytes. You can modify these examples to base the query on some other property of the FileInfo object.

C#

```
// Return the FileInfo object for the largest file
// by sorting and selecting from beginning of list
FileInfo longestFile = (from file in fileList
                        let fileInfo = new FileInfo(file)
                        where fileInfo.Length > 0
                        orderby fileInfo.Length descending
                        select fileInfo
                        ).First();

Console.WriteLine($"The largest file under {startFolder} is
{longestFile.FullName} with a length of {longestFile.Length} bytes");

//Return the FileInfo of the smallest file
FileInfo smallestFile = (from file in fileList
                         let fileInfo = new FileInfo(file)
                         where fileInfo.Length > 0
                         orderby fileInfo.Length ascending
                         select fileInfo
                        ).First();

Console.WriteLine($"The smallest file under {startFolder} is
{smallestFile.FullName} with a length of {smallestFile.Length} bytes");

//Return the FileInfos for the 10 largest files
var queryTenLargest = (from file in fileList
                       let fileInfo = new FileInfo(file)
                       let len = fileInfo.Length
                       orderby len descending
                       select fileInfo
                      ).Take(10);

Console.WriteLine($"The 10 largest files under {startFolder} are:");

foreach (var v in queryTenLargest)
{
```

```
    Console.WriteLine($"{v.FullName}: {v.Length} bytes");
}

// Group the files according to their size, leaving out
// files that are less than 200000 bytes.
var querySizeGroups = from file in fileList
                      let fileInfo = new FileInfo(file)
                      let len = fileInfo.Length
                      where len > 0
                      group fileInfo by (len / 100000) into fileGroup
                      where fileGroup.Key >= 2
                      orderby fileGroup.Key descending
                      select fileGroup;

foreach (var filegroup in querySizeGroups)
{
    Console.WriteLine($"{filegroup.Key}00000");
    foreach (var item in filegroup)
    {
        Console.WriteLine($"\t{item.Name}: {item.Length}");
    }
}
```

To return one or more complete FileInfo objects, the query first must examine each one in the data source, and then sort them by the value of their Length property. Then it can return the single one or the sequence with the greatest lengths. Use First to return the first element in a list. Use Take to return the first n number of elements. Specify a descending sort order to put the smallest elements at the start of the list.

How to query for duplicate files in a directory tree

Sometimes files that have the same name can be located in more than one folder. This example shows how to query for such duplicate file names under a specified root folder. The second example shows how to query for files whose size and LastWrite times also match.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);
```

```
// used in WriteLine to keep the lines shorter
int charsToSkip = startFolder.Length;

// var can be used for convenience with groups.
var queryDupNames = from file in fileList
                    group file.FullName.Substring(charsToSkip) by file.Name
into fileGroup
                    where fileGroup.Count() > 1
                    select fileGroup;

foreach (var queryDup in queryDupNames.Take(20))
{
    Console.WriteLine($"Filename = {(queryDup.Key.ToString() == string.Empty
? "[none]" : queryDup.Key.ToString())}");

    foreach (var fileName in queryDup.Take(10))
    {
        Console.WriteLine($"\t{fileName}");
    }
}
```

The first query uses a key to determine a match. It finds files that have the same name but whose contents might be different. The second query uses a compound key to match against three properties of the FileInfo object. This query is much more likely to find files that have the same name and similar or identical content.

C#

```
    string startFolder = """C:\Program Files\dotnet\sdk""";
    // Or
    // string startFolder = "/usr/local/share/dotnet/sdk";

    // Make the lines shorter for the console display
    int charsToSkip = startFolder.Length;

    // Take a snapshot of the file system.
    DirectoryInfo dir = new DirectoryInfo(startFolder);
    IEnumerable<FileInfo> fileList = dir.GetFiles("*.*",
SearchOption.AllDirectories);

    // Note the use of a compound key. Files that match
    // all three properties belong to the same group.
    // A named type is used to enable the query to be
    // passed to another method. Anonymous types can also be used
    // for composite keys but cannot be passed across method boundaries
    //
    var queryDupFiles = from file in fileList
                        group file.FullName.Substring(charsToSkip) by
                        (Name: file.Name, LastWriteTime: file.LastWriteTime,
Length: file.Length )
                        into fileGroup
                        where fileGroup.Count() > 1
```

```
                        select fileGroup;

    foreach (var queryDup in queryDupFiles.Take(20))
    {
        Console.WriteLine($"Filename = {(queryDup.Key.ToString() ==
string.Empty ? "[none]" : queryDup.Key.ToString())}");

        foreach (var fileName in queryDup)
        {
            Console.WriteLine($"\t{fileName}");
        }
    }
}
```

How to query the contents of text files in a folder

This example shows how to query over all the files in a specified directory tree, open each file, and inspect its contents. This type of technique could be used to create indexes or reverse indexes of the contents of a directory tree. A simple string search is performed in this example. However, more complex types of pattern matching can be performed with a regular expression.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

DirectoryInfo dir = new DirectoryInfo(startFolder);

var fileList = dir.GetFiles("*.*", SearchOption.AllDirectories);

string searchTerm = "change";

var queryMatchingFiles = from file in fileList
                         where file.Extension == ".txt"
                         let fileText = File.ReadAllText(file.FullName)
                         where fileText.Contains(searchTerm)
                         select file.FullName;

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");
foreach (string filename in queryMatchingFiles)
{
    Console.WriteLine(filename);
}
```

How to compare the contents of two folders

This example demonstrates three ways to compare two file listings:

- By querying for a Boolean value that specifies whether the two file lists are identical.
- By querying for the intersection to retrieve the files that are in both folders.
- By querying for the set difference to retrieve the files that are in one folder but not the other.

The techniques shown here can be adapted to compare sequences of objects of any type.

The `FileComparer` class shown here demonstrates how to use a custom comparer class together with the Standard Query Operators. The class isn't intended for use in real-world scenarios. It just uses the name and length in bytes of each file to determine whether the contents of each folder are identical or not. In a real-world scenario, you should modify this comparer to perform a more rigorous equality check.

C#

```
// This implementation defines a very simple comparison
// between two FileInfo objects. It only compares the name
// of the files being compared and their length in bytes.
class FileCompare : IEqualityComparer<FileInfo>
{
    public bool Equals(FileInfo? f1, FileInfo? f2)
    {
        return (f1?.Name == f2?.Name &&
                f1?.Length == f2?.Length);
    }

    // Return a hash that reflects the comparison criteria. According to the
    // rules for IEqualityComparer<T>, if Equals is true, then the hash
codes must
    // also be equal. Because equality as defined here is a simple value
equality, not
    // reference identity, it is possible that two or more objects will
produce the same
    // hash code.
    public int GetHashCode(FileInfo fi)
    {
        string s = $"{fi.Name}{fi.Length}";
        return s.GetHashCode();
    }
}

public static void CompareDirectories()
{
```

```
    string pathA = """C:\Program Files\dotnet\sdk\8.0.104""";
    string pathB = """C:\Program Files\dotnet\sdk\8.0.204""";

    DirectoryInfo dir1 = new DirectoryInfo(pathA);
    DirectoryInfo dir2 = new DirectoryInfo(pathB);

    IEnumerable<FileInfo> list1 = dir1.GetFiles("*.*",
SearchOption.AllDirectories);
    IEnumerable<FileInfo> list2 = dir2.GetFiles("*.*",
SearchOption.AllDirectories);

    //A custom file comparer defined below
    FileCompare myFileCompare = new FileCompare();

    // This query determines whether the two folders contain
    // identical file lists, based on the custom file comparer
    // that is defined in the FileCompare class.
    // The query executes immediately because it returns a bool.
    bool areIdentical = list1.SequenceEqual(list2, myFileCompare);

    if (areIdentical == true)
    {
        Console.WriteLine("the two folders are the same");
    }
    else
    {
        Console.WriteLine("The two folders are not the same");
    }

    // Find the common files. It produces a sequence and doesn't
    // execute until the foreach statement.
    var queryCommonFiles = list1.Intersect(list2, myFileCompare);

    if (queryCommonFiles.Any())
    {
        Console.WriteLine($"The following files are in both folders (total
number = {queryCommonFiles.Count()}):");
        foreach (var v in queryCommonFiles.Take(10))
        {
            Console.WriteLine(v.Name); //shows which items end up in result
list
        }
    }
    else
    {
        Console.WriteLine("There are no common files in the two folders.");
    }

    // Find the set difference between the two folders.
    var queryList1Only = (from file in list1
                          select file)
                          .Except(list2, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list1 but not list2
```

```
(total number = {queryList1Only.Count()}):");
    foreach (var v in queryList1Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }

    var queryList2Only = (from file in list2
                          select file)
                          .Except(list1, myFileCompare);

    Console.WriteLine();
    Console.WriteLine($"The following files are in list2 but not list1
(total number = {queryList2Only.Count()}:");
    foreach (var v in queryList2Only.Take(10))
    {
        Console.WriteLine(v.FullName);
    }
}
```

How to reorder the fields of a delimited file

A comma-separated value (CSV) file is a text file that is often used to store spreadsheet data or other tabular data represented by rows and columns. By using the Split method to separate the fields, it's easy to query and manipulate CSV files using LINQ. In fact, the same technique can be used to reorder the parts of any structured line of text; it isn't limited to CSV files.

In the following example, assume that the three columns represent students' "family name," "first name", and "ID." The fields are in alphabetical order based on the students' family names. The query produces a new sequence in which the ID column appears first, followed by a second column that combines the student's first name and family name. The lines are reordered according to the ID field. The results are saved into a new file and the original data isn't modified. The following text shows the contents of the *spreadsheet1.csv* file used in the following example:

txt

```
Adams,Terry,120
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Cesar,114
Garcia,Debra,115
Garcia,Hugo,118
Mortensen,Sven,113
O'Donnell,Claire,112
Omelchenko,Svetlana,111
Tucker,Lance,119
```

```
Tucker,Michael,122
Zabokritski,Eugene,121
```

The following code reads the source file and rearranges each column in the CSV file to rearrange the order of the columns:

C#

```
string[] lines = File.ReadAllLines("spreadsheet1.csv");

// Create the query. Put field 2 first, then
// reverse and combine fields 0 and 1 from the old field
IEnumerable<string> query = from line in lines
                            let fields = line.Split(',')
                            orderby fields[2]
                            select $"{fields[2]}, {fields[1]} {fields[0]}";

File.WriteAllLines("spreadsheet2.csv", query.ToArray());

/* Output to spreadsheet2.csv:
111, Svetlana Omelchenko
112, Claire O'Donnell
113, Sven Mortensen
114, Cesar Garcia
115, Debra Garcia
116, Fadi Fakhouri
117, Hanying Feng
118, Hugo Garcia
119, Lance Tucker
120, Terry Adams
121, Eugene Zabokritski
122, Michael Tucker
*/
```

How to split a file into many files by using groups

This example shows one way to merge the contents of two files and then create a set of new files that organize the data in a new way. The query uses the contents of two files. The following text shows the contents of the first file, *names1.txt*:

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
```

```
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

The second file, *names2.txt*, contains a different set of names, some of which are in common with the first set:

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

The following code queries both files, takes the union of both files, then writes a new file for each group, defined by the first letter of the family name:

C#

```
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

// Concatenate and remove duplicate names
var mergeQuery = fileA.Union(fileB);

// Group the names by the first letter in the last name.
var groupQuery = from name in mergeQuery
                 let n = name.Split(',')[0]
                 group name by n[0] into g
                 orderby g.Key
                 select g;

foreach (var g in groupQuery)
{
    string fileName = $"testFile_{g.Key}.txt";

    Console.WriteLine(g.Key);

    using StreamWriter sw = new StreamWriter(fileName);
    foreach (var item in g)
    {
        sw.WriteLine(item);
        // Output to console for example purposes.
```

```
            Console.WriteLine($"   {item}");
        }
    }
    /* Output:
        A
           Aw, Kam Foo
        B
           Bankov, Peter
           Beebe, Ann
        E
           El Yassir, Mehdi
        G
           Garcia, Hugo
           Guy, Wey Yuan
           Garcia, Debra
           Gilchrist, Beth
           Giakoumakis, Leo
        H
           Holm, Michael
        L
           Liu, Jinghao
        M
           Myrcha, Jacek
           McLin, Nkenge
        N
           Noriega, Fabricio
        P
           Potra, Cristina
        T
           Toyoshima, Tim
     */
```

How to join content from dissimilar files

This example shows how to join data from two comma-delimited files that share a common value that is used as a matching key. This technique can be useful if you have to combine data from two spreadsheets, or from a spreadsheet and from a file that has another format, into a new file. You can modify the example to work with any kind of structured text.

The following text shows the contents of *scores.csv*. The file represents spreadsheet data. Column 1 is the student's ID, and columns 2 through 5 are test scores.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
```

```
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

The following text shows the contents of *names.csv*. The file represents a spreadsheet that contains the student's family name, first name, and student ID.

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

Join content from dissimilar files that contain related information. File *names.csv* contains the student name plus an ID number. File *scores.csv* contains the ID and a set of four test scores. The following query joins the scores to the student names by using ID as a matching key. The code is shown in the following example:

C#

```
string[] names = File.ReadAllLines(@"names.csv");
string[] scores = File.ReadAllLines(@"scores.csv");

var scoreQuery = from name in names
                 let nameFields = name.Split(',')
                 from id in scores
                 let scoreFields = id.Split(',')
                 where Convert.ToInt32(nameFields[2]) ==
Convert.ToInt32(scoreFields[0])
                 select $"{nameFields[0]},{scoreFields[1]},
{scoreFields[2]},{scoreFields[3]},{scoreFields[4]}";

Console.WriteLine("\r\nMerge two spreadsheets:");
foreach (string item in scoreQuery)
{
    Console.WriteLine(item);
}
```

```
Console.WriteLine($"{scoreQuery.Count()} total names in list");
/* Output:
Merge two spreadsheets:
Omelchenko, 97, 92, 81, 60
O'Donnell, 75, 84, 91, 39
Mortensen, 88, 94, 65, 91
Garcia, 97, 89, 85, 82
Garcia, 35, 72, 91, 70
Fakhouri, 99, 86, 90, 94
Feng, 93, 92, 80, 87
Garcia, 92, 90, 83, 78
Tucker, 68, 79, 88, 92
Adams, 99, 82, 81, 79
Zabokritski, 96, 85, 91, 60
Tucker, 94, 92, 91, 91
12 total names in list
 */
```

How to compute column values in a CSV text file

This example shows how to perform aggregate computations such as Sum, Average, Min, and Max on the columns of a .csv file. The example principles that are shown here can be applied to other types of structured text.

The following text shows the contents of *scores.csv*. Assume that the first column represents a student ID, and subsequent columns represent scores from four exams.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

The following text shows how to use the Split method to convert each line of text into an array. Each array element represents a column. Finally, the text in each column is converted to its numeric representation.

```
```

C#

```
public class SumColumns
{
    public static void SumCSVColumns(string fileName)
    {
        string[] lines = File.ReadAllLines(fileName);

        // Specifies the column to compute.
        int exam = 3;

        // Spreadsheet format:
        // Student ID    Exam#1  Exam#2  Exam#3  Exam#4
        // 111,          97,     92,     81,     60

        // Add one to exam to skip over the first column,
        // which holds the student ID.
        SingleColumn(lines, exam + 1);
        Console.WriteLine();
        MultiColumns(lines);
    }

    static void SingleColumn(IEnumerable<string> strs, int examNum)
    {
        Console.WriteLine("Single Column Query:");

        // Parameter examNum specifies the column to
        // run the calculations on. This value could be
        // passed in dynamically at run time.

        // Variable columnQuery is an IEnumerable<int>.
        // The following query performs two steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) convert the element at position examNum to an int
        //    and select it.
        var columnQuery = from line in strs
                          let elements = line.Split(',')
                          select Convert.ToInt32(elements[examNum]);

        // Execute the query and cache the results to improve
        // performance. This is helpful only with very large files.
        var results = columnQuery.ToList();

        // Perform aggregate calculations Average, Max, and
        // Min on the column specified by examNum.
        double average = results.Average();
        int max = results.Max();
        int min = results.Min();

        Console.WriteLine($"Exam #{examNum}: Average:{average:##.##} High
Score:{max} Low Score:{min}");
    }
```

```
    static void MultiColumns(IEnumerable<string> strs)
    {
        Console.WriteLine("Multi Column Query:");

        // Create a query, multiColQuery. Explicit typing is used
        // to make clear that, when executed, multiColQuery produces
        // nested sequences. However, you get the same results by
        // using 'var'.

        // The multiColQuery query performs the following steps:
        // 1) use Split to break each row (a string) into an array
        //    of strings,
        // 2) use Skip to skip the "Student ID" column, and store the
        //    rest of the row in scores.
        // 3) convert each score in the current row from a string to
        //    an int, and select that entire sequence as one row
        //    in the results.
        var multiColQuery = from line in strs
                            let elements = line.Split(',')
                            let scores = elements.Skip(1)
                            select (from str in scores
                                    select Convert.ToInt32(str));

        // Execute the query and cache the results to improve
        // performance.
        // ToArray could be used instead of ToList.
        var results = multiColQuery.ToList();

        // Find out how many columns you have in results.
        int columnCount = results[0].Count();

        // Perform aggregate calculations Average, Max, and
        // Min on each column.
        // Perform one iteration of the loop for each column
        // of scores.
        // You can use a for loop instead of a foreach loop
        // because you already executed the multiColQuery
        // query by calling ToList.
        for (int column = 0; column < columnCount; column++)
        {
            var results2 = from row in results
                           select row.ElementAt(column);
            double average = results2.Average();
            int max = results2.Max();
            int min = results2.Min();

            // Add one to column because the first exam is Exam #1,
            // not Exam #0.
            Console.WriteLine($"Exam #{column + 1} Average: {average:##.##}
High Score: {max} Low Score: {min}");
        }
    }
}
/* Output:
    Single Column Query:
```

```
    Exam #4: Average:76.92 High Score:94 Low Score:39

    Multi Column Query:
    Exam #1 Average: 86.08 High Score: 99 Low Score: 35
    Exam #2 Average: 86.42 High Score: 94 Low Score: 72
    Exam #3 Average: 84.75 High Score: 91 Low Score: 65
    Exam #4 Average: 76.92 High Score: 94 Low Score: 39
*/
```

If your file is a tab-separated file, just update the argument in the `Split` method to `\t`.

How to: Use LINQ to query strings

Article • 04/25/2024

Strings are stored as a sequence of characters. As a sequence of characters, they can be queried using LINQ. In this article, there are several example queries that query strings for different characters or words, filter strings, or mix queries with regular expressions.

How to query for characters in a string

The following example queries a string to determine the number of numeric digits it contains.

C#

```
string aString = "ABCDE99F-J74-12-89A";

// Select only those characters that are numbers
var stringQuery = from ch in aString
                  where Char.IsDigit(ch)
                  select ch;

// Execute the query
foreach (char c in stringQuery)
    Console.Write(c + " ");

// Call the Count method on the existing query.
int count = stringQuery.Count();
Console.WriteLine($"Count = {count}");

// Select all characters before the first '-'
var stringQuery2 = aString.TakeWhile(c => c != '-');

// Execute the second query
foreach (char c in stringQuery2)
    Console.Write(c);
/* Output:
  Output: 9 9 7 4 1 2 8 9
  Count = 8
  ABCDE99F
*/
```

The preceding query shows how you can treat a string as a sequence of characters.

How to count occurrences of a word in a string

The following example shows how to use a LINQ query to count the occurrences of a specified word in a string. To perform the count, first the Split method is called to create an array of words. There's a performance cost to the Split method. If the only operation on the string is to count the words, consider using the Matches or IndexOf methods instead.

C#

```
string text = """
    Historically, the world of data and the world of objects
    have not been well integrated. Programmers work in C# or Visual Basic
    and also in SQL or XQuery. On the one side are concepts such as classes,
    objects, fields, inheritance, and .NET APIs. On the other side
    are tables, columns, rows, nodes, and separate languages for dealing
with
    them. Data types often require translation between the two worlds; there
are
    different standard functions. Because the object world has no notion of
query, a
    query can only be represented as a string without compile-time type
checking or
    IntelliSense support in the IDE. Transferring data from SQL tables or
XML trees to
    objects in memory is often tedious and error-prone.
    """;

string searchTerm = "data";

//Convert the string into an array of words
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
string[] source = text.Split(separators,
StringSplitOptions.RemoveEmptyEntries);

// Create the query.  Use the InvariantCultureIgnoreCase comparison to match
"data" and "Data"
var matchQuery = from word in source
                 where word.Equals(searchTerm,
StringComparison.InvariantCultureIgnoreCase)
                 select word;

// Count the matches, which executes the query.
int wordCount = matchQuery.Count();
Console.WriteLine($"""{wordCount} occurrences(s) of the search term "
{searchTerm}" were found.""");
/* Output:
   3 occurrences(s) of the search term "data" were found.
*/
```

The preceding query shows how you can view strings as a sequence of words, after splitting a string into a sequence of words.

How to sort or filter text data by any word or field

The following example shows how to sort lines of structured text, such as comma-separated values, by any field in the line. The field can be dynamically specified at run time. Assume that the fields in *scores.csv* represent a student's ID number, followed by a series of four test scores:

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

The following query sorts the lines based on the score of the first exam, stored in the second column:

C#

```
// Create an IEnumerable data source
string[] scores = File.ReadAllLines("scores.csv");

// Change this to any value from 0 to 4.
int sortField = 1;

Console.WriteLine($"Sorted highest to lowest by field [{sortField}]:");

// Split the string and sort on field[num]
var scoreQuery = from line in scores
                 let fields = line.Split(',')
                 orderby fields[sortField] descending
                 select line;

foreach (string str in scoreQuery)
{
    Console.WriteLine(str);
}
/* Output (if sortField == 1):
   Sorted highest to lowest by field [1]:
    116, 99, 86, 90, 94
```

```
    120, 99, 82, 81, 79
    111, 97, 92, 81, 60
    114, 97, 89, 85, 82
    121, 96, 85, 91, 60
    122, 94, 92, 91, 91
    117, 93, 92, 80, 87
    118, 92, 90, 83, 78
    113, 88, 94, 65, 91
    112, 75, 84, 91, 39
    119, 68, 79, 88, 92
    115, 35, 72, 91, 70
 */
```

The preceding query shows how you can manipulate strings by splitting them into fields, and querying the individual fields.

How to query for sentences with specific words

The following example shows how to find sentences in a text file that contain matches for each of a specified set of words. Although the array of search terms is hard-coded, it could also be populated dynamically at run time. The query returns the sentences that contain the words "Historically," "data," and "integrated."

C#

```
string text = """
Historically, the world of data and the world of objects
have not been well integrated. Programmers work in C# or Visual Basic
and also in SQL or XQuery. On the one side are concepts such as classes,
objects, fields, inheritance, and .NET APIs. On the other side
are tables, columns, rows, nodes, and separate languages for dealing with
them. Data types often require translation between the two worlds; there are
different standard functions. Because the object world has no notion of
query, a
query can only be represented as a string without compile-time type checking
or
IntelliSense support in the IDE. Transferring data from SQL tables or XML
trees to
objects in memory is often tedious and error-prone.
""";

// Split the text block into an array of sentences.
string[] sentences = text.Split(['.', '?', '!']);

// Define the search terms. This list could also be dynamically populated at
run time.
string[] wordsToMatch = [ "Historically", "data", "integrated" ];

// Find sentences that contain all the terms in the wordsToMatch array.
// Note that the number of terms to match is not specified at compile time.
```

```
char[] separators = ['.', '?', '!', ' ', ';', ':', ','];
var sentenceQuery = from sentence in sentences
                    let w =
sentence.Split(separators,StringSplitOptions.RemoveEmptyEntries)
                    where w.Distinct().Intersect(wordsToMatch).Count() ==
wordsToMatch.Count()
                    select sentence;

foreach (string str in sentenceQuery)
{
    Console.WriteLine(str);
}
/* Output:
Historically, the world of data and the world of objects have not been well
integrated
*/
```

The query first splits the text into sentences, and then splits each sentence into an array of strings that hold each word. For each of these arrays, the Distinct method removes all duplicate words, and then the query performs an Intersect operation on the word array and the `wordsToMatch` array. If the count of the intersection is the same as the count of the `wordsToMatch` array, all words were found in the words and the original sentence is returned.

The call to Split uses punctuation marks as separators in order to remove them from the string. If you didn't do remove punctuation, for example you could have a string "Historically," that wouldn't match "Historically" in the `wordsToMatch` array. You might have to use extra separators, depending on the types of punctuation found in the source text.

How to combine LINQ queries with regular expressions

The following example shows how to use the Regex class to create a regular expression for more complex matching in text strings. The LINQ query makes it easy to filter on exactly the files that you want to search with the regular expression, and to shape the results.

C#

```
string startFolder = """C:\Program Files\dotnet\sdk""";
// Or
// string startFolder = "/usr/local/share/dotnet/sdk";

// Take a snapshot of the file system.
var fileList = from file in Directory.GetFiles(startFolder, "*.*",
```

```
SearchOption.AllDirectories)
                let fileInfo = new FileInfo(file)
                select fileInfo;

// Create the regular expression to find all things "Visual".
System.Text.RegularExpressions.Regex searchTerm =
    new System.Text.RegularExpressions.Regex(@"microsoft.net.
(sdk|workload)");

// Search the contents of each .htm file.
// Remove the where clause to find even more matchedValues!
// This query produces a list of files where a match
// was found, and a list of the matchedValues in that file.
// Note: Explicit typing of "Match" in select clause.
// This is required because MatchCollection is not a
// generic IEnumerable collection.
var queryMatchingFiles =
    from file in fileList
    where file.Extension == ".txt"
    let fileText = File.ReadAllText(file.FullName)
    let matches = searchTerm.Matches(fileText)
    where matches.Count > 0
    select new
    {
        name = file.FullName,
        matchedValues = from System.Text.RegularExpressions.Match match in
matches
                        select match.Value
    };

// Execute the query.
Console.WriteLine($"""The term "{searchTerm}" was found in:""");

foreach (var v in queryMatchingFiles)
{
    // Trim the path a bit, then write
    // the file name in which a match was found.
    string s = v.name.Substring(startFolder.Length - 1);
    Console.WriteLine(s);

    // For this file, write out all the matching strings
    foreach (var v2 in v.matchedValues)
    {
        Console.WriteLine($"  {v2}");
    }
}
```

You can also query the MatchCollection object returned by a `RegEx` search. Only the value of each match is produced in the results. However, it's also possible to use LINQ to perform all kinds of filtering, sorting, and grouping on that collection. Because MatchCollection is a nongeneric IEnumerable collection, you have to explicitly state the type of the range variable in the query.

LINQ and collections

Article • 04/25/2024

Most collections model a *sequence* of elements. You can use LINQ to query any collection type. Other LINQ methods find elements in a collection, compute values from the elements in a collection, or modify the collection or its elements. These examples help you learn about LINQ methods and how you can use them with your collections, or other data sources.

How to find the set difference between two lists

This example shows how to use LINQ to compare two lists of strings and output those lines that are in first collection, but not in the second. The first collection of names is stored in the file *names1.txt*:

txt

```
Bankov, Peter
Holm, Michael
Garcia, Hugo
Potra, Cristina
Noriega, Fabricio
Aw, Kam Foo
Beebe, Ann
Toyoshima, Tim
Guy, Wey Yuan
Garcia, Debra
```

The second collection of names is stored in the file *names2.txt*. Some names appear in both sequences.

txt

```
Liu, Jinghao
Bankov, Peter
Holm, Michael
Garcia, Hugo
Beebe, Ann
Gilchrist, Beth
Myrcha, Jacek
Giakoumakis, Leo
McLin, Nkenge
El Yassir, Mehdi
```

The following code shows how you can use the Enumerable.Except method to find elements in the first list that aren't in the second list:

C#

```
// Create the IEnumerable data sources.
string[] names1 = File.ReadAllLines("names1.txt");
string[] names2 = File.ReadAllLines("names2.txt");

// Create the query. Note that method syntax must be used here.
var differenceQuery = names1.Except(names2);

// Execute the query.
Console.WriteLine("The following lines are in names1.txt but not
names2.txt");
foreach (string s in differenceQuery)
    Console.WriteLine(s);
/* Output:
 The following lines are in names1.txt but not names2.txt
 Potra, Cristina
 Noriega, Fabricio
 Aw, Kam Foo
 Toyoshima, Tim
 Guy, Wey Yuan
 Garcia, Debra
 */
```

Some types of query operations, such as Except, Distinct, Union, and Concat, can only be expressed in method-based syntax.

How to combine and compare string collections

This example shows how to merge files that contain lines of text and then sort the results. Specifically, it shows how to perform a concatenation, a union, and an intersection on the two sets of text lines. It uses the same two text files shows in the preceding example. The code shows examples of the Enumerable.Concat, Enumerable.Union, and Enumerable.Except.

C#

```
//Put text files in your solution folder
string[] fileA = File.ReadAllLines("names1.txt");
string[] fileB = File.ReadAllLines("names2.txt");

//Simple concatenation and sort. Duplicates are preserved.
var concatQuery = fileA.Concat(fileB).OrderBy(s => s);
```

```
// Pass the query variable to another function for execution.
OutputQueryResults(concatQuery, "Simple concatenate and sort. Duplicates are
preserved:");

// Concatenate and remove duplicate names based on
// default string comparer.
var uniqueNamesQuery = fileA.Union(fileB).OrderBy(s => s);
OutputQueryResults(uniqueNamesQuery, "Union removes duplicate names:");

// Find the names that occur in both files (based on
// default string comparer).
var commonNamesQuery = fileA.Intersect(fileB);
OutputQueryResults(commonNamesQuery, "Merge based on intersect:");

// Find the matching fields in each list. Merge the two
// results by using Concat, and then
// sort using the default string comparer.
string nameMatch = "Garcia";

var tempQuery1 = from name in fileA
                 let n = name.Split(',')
                 where n[0] == nameMatch
                 select name;

var tempQuery2 = from name2 in fileB
                 let n2 = name2.Split(',')
                 where n2[0] == nameMatch
                 select name2;

var nameMatchQuery = tempQuery1.Concat(tempQuery2).OrderBy(s => s);
OutputQueryResults(nameMatchQuery, $"""Concat based on partial name match "
{nameMatch}":""");

static void OutputQueryResults(IEnumerable<string> query, string message)
{
    Console.WriteLine(Environment.NewLine + message);
    foreach (string item in query)
    {
        Console.WriteLine(item);
    }
    Console.WriteLine($"{query.Count()} total names in list");
}
/* Output:
    Simple concatenate and sort. Duplicates are preserved:
    Aw, Kam Foo
    Bankov, Peter
    Bankov, Peter
    Beebe, Ann
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
```

```
    Guy, Wey Yuan
    Holm, Michael
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    20 total names in list

    Union removes duplicate names:
    Aw, Kam Foo
    Bankov, Peter
    Beebe, Ann
    El Yassir, Mehdi
    Garcia, Debra
    Garcia, Hugo
    Giakoumakis, Leo
    Gilchrist, Beth
    Guy, Wey Yuan
    Holm, Michael
    Liu, Jinghao
    McLin, Nkenge
    Myrcha, Jacek
    Noriega, Fabricio
    Potra, Cristina
    Toyoshima, Tim
    16 total names in list

    Merge based on intersect:
    Bankov, Peter
    Holm, Michael
    Garcia, Hugo
    Beebe, Ann
    4 total names in list

    Concat based on partial name match "Garcia":
    Garcia, Debra
    Garcia, Hugo
    Garcia, Hugo
    3 total names in list
*/
```

How to populate object collections from multiple sources

This example shows how to merge data from different sources into a sequence of new types.

> **Note**
>
> Don't try to join in-memory data or data in the file system with data that is still in a database. Such cross-domain joins can yield undefined results because of different ways in which join operations might be defined for database queries and other types of sources. Additionally, there is a risk that such an operation could cause an out-of-memory exception if the amount of data in the database is large enough. To join data from a database to in-memory data, first call `ToList` or `ToArray` on the database query, and then perform the join on the returned collection.

This example uses two files. The first, *names.csv*, contains student names and student IDs.

txt

```
Omelchenko,Svetlana,111
O'Donnell,Claire,112
Mortensen,Sven,113
Garcia,Cesar,114
Garcia,Debra,115
Fakhouri,Fadi,116
Feng,Hanying,117
Garcia,Hugo,118
Tucker,Lance,119
Adams,Terry,120
Zabokritski,Eugene,121
Tucker,Michael,122
```

The second, *scores.csv*, contains student IDs in the first column, followed by exam scores.

txt

```
111, 97, 92, 81, 60
112, 75, 84, 91, 39
113, 88, 94, 65, 91
114, 97, 89, 85, 82
115, 35, 72, 91, 70
116, 99, 86, 90, 94
117, 93, 92, 80, 87
118, 92, 90, 83, 78
119, 68, 79, 88, 92
120, 99, 82, 81, 79
121, 96, 85, 91, 60
122, 94, 92, 91, 91
```

The following example shows how to use a named record `Student` to store merged data from two in-memory collections of strings that simulate spreadsheet data in .csv format. The ID is used as the key to map students to their scores.

C#

```
// Each line of names.csv consists of a last name, a first name, and an
// ID number, separated by commas. For example, Omelchenko,Svetlana,111
string[] names = File.ReadAllLines("names.csv");

// Each line of scores.csv consists of an ID number and four test
// scores, separated by commas. For example, 111, 97, 92, 81, 60
string[] scores = File.ReadAllLines("scores.csv");

// Merge the data sources using a named type.
// var could be used instead of an explicit type. Note the dynamic
// creation of a list of ints for the ExamScores member. The first item
// is skipped in the split string because it is the student ID,
// not an exam score.
IEnumerable<Student> queryNamesScores = from nameLine in names
                                        let splitName = nameLine.Split(',')
                                        from scoreLine in scores
                                        let splitScoreLine =
scoreLine.Split(',')
                                        where Convert.ToInt32(splitName[2])
== Convert.ToInt32(splitScoreLine[0])
                                        select new Student
                                        (
                                            FirstName: splitName[0],
                                            LastName: splitName[1],
                                            ID:
Convert.ToInt32(splitName[2]),
                                            ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                                         select
Convert.ToInt32(scoreAsText)
                                                        ).ToArray()
                                        );

// Optional. Store the newly created student objects in memory
// for faster access in future queries. This could be useful with
// very large data files.
List<Student> students = queryNamesScores.ToList();

// Display each student's name and exam score average.
foreach (var student in students)
{
    Console.WriteLine($"The average score of {student.FirstName}
{student.LastName} is {student.ExamScores.Average()}.");
}
/* Output:
The average score of Omelchenko Svetlana is 82.5.
The average score of O'Donnell Claire is 72.25.
```

```
The average score of Mortensen Sven is 84.5.
The average score of Garcia Cesar is 88.25.
The average score of Garcia Debra is 67.
The average score of Fakhouri Fadi is 92.25.
The average score of Feng Hanying is 88.
The average score of Garcia Hugo is 85.75.
The average score of Tucker Lance is 81.75.
The average score of Adams Terry is 85.25.
The average score of Zabokritski Eugene is 83.
The average score of Tucker Michael is 92.
*/
```

In the select clause, each new `Student` object is initialized from the data in the two sources.

If you don't have to store the results of a query, tuples or anonymous types can be more convenient than named types. The following example executes the same task as the previous example, but uses tuples instead of named types:

C#

```
// Merge the data sources by using an anonymous type.
// Note the dynamic creation of a list of ints for the
// ExamScores member. We skip 1 because the first string
// in the array is the student ID, not an exam score.
var queryNamesScores2 = from nameLine in names
                        let splitName = nameLine.Split(',')
                        from scoreLine in scores
                        let splitScoreLine = scoreLine.Split(',')
                        where Convert.ToInt32(splitName[2]) ==
Convert.ToInt32(splitScoreLine[0])
                        select (FirstName: splitName[0],
                                LastName: splitName[1],
                                ExamScores: (from scoreAsText in
splitScoreLine.Skip(1)
                                             select
Convert.ToInt32(scoreAsText))
                                             .ToList()
                                );

// Display each student's name and exam score average.
foreach (var student in queryNamesScores2)
{
    Console.WriteLine($"The average score of {student.FirstName}
{student.LastName} is {student.ExamScores.Average()}.");
}
```

How to query an ArrayList with LINQ

When using LINQ to query nongeneric IEnumerable collections such as ArrayList, you must explicitly declare the type of the range variable to reflect the specific type of the objects in the collection. If you have an ArrayList of `Student` objects, your from clause should look like this:

C#

```
var query = from Student s in arrList
//...
```

By specifying the type of the range variable, you're casting each item in the ArrayList to a `Student`.

The use of an explicitly typed range variable in a query expression is equivalent to calling the Cast method. Cast throws an exception if the specified cast can't be performed. Cast and OfType are the two Standard Query Operator methods that operate on nongeneric IEnumerable types. For more information, see Type Relationships in LINQ Query Operations. The following example shows a query over an ArrayList.

C#

```
ArrayList arrList = new ArrayList();
arrList.Add(
    new Student
    (
        FirstName: "Svetlana",
        LastName: "Omelchenko",
        ExamScores: new int[] { 98, 92, 81, 60 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Claire",
        LastName: "O'Donnell",
        ExamScores: new int[] { 75, 84, 91, 39 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Sven",
        LastName: "Mortensen",
        ExamScores: new int[] { 88, 94, 65, 91 }
    ));
arrList.Add(
    new Student
    (
        FirstName: "Cesar",
        LastName: "Garcia",
        ExamScores: new int[] { 97, 89, 85, 82 }
```

```
    ));

var query = from Student student in arrList
            where student.ExamScores[0] > 95
            select student;

foreach (Student s in query)
    Console.WriteLine(s.LastName + ": " + s.ExamScores[0]);
```

Write your own extensions to LINQ

All LINQ based methods follow one of two similar patterns. They take an enumerable sequence. They return either a different sequence or a single value. The consistency of the shape enables you to extend LINQ by writing methods with a similar shape. In fact, the .NET libraries gained new methods in many .NET releases since LINQ was first introduced. In this article, you see examples of extending LINQ by writing your own methods that follow the same pattern.

Add custom methods for LINQ queries

Extend the set of methods that you use for LINQ queries by adding extension methods to the IEnumerable<T> interface. For example, in addition to the standard average or maximum operations, you create a custom aggregate method to compute a single value from a sequence of values. You also create a method that works as a custom filter or a specific data transform for a sequence of values and returns a new sequence. Examples of such methods are Distinct, Skip, and Reverse.

When you extend the IEnumerable<T> interface, you can apply your custom methods to any enumerable collection. For more information, see Extension members.

An *aggregate* method computes a single value from a set of values. LINQ provides several aggregate methods, including Average, Min, and Max. Create your own aggregate method by adding an extension method to the IEnumerable<T> interface.

Beginning in C# 14, you can declare an *extension block* to contain multiple extension members. Declare an extension block with the keyword `extension` followed by the receiver parameter in parentheses. The following code example shows how to create an extension method called `Median` in an extension block. The method computes a median for a sequence of numbers of type `double`.

C#

```
extension(IEnumerable<double>? source)
{
    public double Median()
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null or
empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();
```

```
        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

You can also add the `this` modifier to a static method to declare an *extension method*. The following code shows the equivalent `Median` extension method:

C#

```
public static class EnumerableExtension
{
    public static double Median(this IEnumerable<double>? source)
    {
        if (source is null || !source.Any())
        {
            throw new InvalidOperationException("Cannot compute median for a null or
empty set.");
        }

        var sortedList =
            source.OrderBy(number => number).ToList();

        int itemIndex = sortedList.Count / 2;

        if (sortedList.Count % 2 == 0)
        {
            // Even number of items.
            return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
        }
        else
        {
            // Odd number of items.
            return sortedList[itemIndex];
        }
    }
}
```

Call either extension method for any enumerable collection in the same way you call other aggregate methods from the IEnumerable<T> interface.

The following code example shows how to use the `Median` method for an array of type `double`.

C#

```
double[] numbers = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query = numbers.Median();

Console.WriteLine($"double: Median = {query}");
// This code produces the following output:
//     double: Median = 4.85
```

Overload your aggregate method so that it accepts sequences of various types. The standard approach is to create an overload for each type. Another approach is to create an overload that takes a generic type and convert it to a specific type by using a delegate. You can also combine both approaches.

Create a specific overload for each type that you want to support. The following code example shows an overload of the `Median` method for the `int` type.

C#

```
// int overload
public static double Median(this IEnumerable<int> source) =>
    (from number in source select (double)number).Median();
```

You can now call the `Median` overloads for both `integer` and `double` types, as shown in the following code:

C#

```
double[] numbers1 = [1.9, 2, 8, 4, 5.7, 6, 7.2, 0];
var query1 = numbers1.Median();

Console.WriteLine($"double: Median = {query1}");

int[] numbers2 = [1, 2, 3, 4, 5];
var query2 = numbers2.Median();

Console.WriteLine($"int: Median = {query2}");
// This code produces the following output:
//     double: Median = 4.85
//     int: Median = 3
```

You can also create an overload that accepts a *generic sequence* of objects. This overload takes a delegate as a parameter and uses it to convert a sequence of objects of a generic type to a specific type.

The following code shows an overload of the `Median` method that takes the Func<T,TResult> delegate as a parameter. This delegate takes an object of generic type T and returns an object of type `double`.

C#

```
// generic overload
public static double Median<T>(
    this IEnumerable<T> numbers, Func<T, double> selector) =>
    (from num in numbers select selector(num)).Median();
```

You can now call the `Median` method for a sequence of objects of any type. If the type doesn't have its own method overload, pass a delegate parameter. In C#, you can use a lambda expression for this purpose. Also, in Visual Basic only, if you use the `Aggregate` or `Group By` clause instead of the method call, you can pass any value or expression that's in the scope this clause.

The following example code shows how to call the `Median` method for an array of integers and an array of strings. For strings, the median for the lengths of strings in the array is calculated. The example shows how to pass the Func<T,TResult> delegate parameter to the `Median` method for each case.

C#

```
int[] numbers3 = [1, 2, 3, 4, 5];

/*
    You can use the num => num lambda expression as a parameter for the Median
method
    so that the compiler will implicitly convert its value to double.
    If there is no implicit conversion, the compiler will display an error message.
*/
var query3 = numbers3.Median(num => num);

Console.WriteLine($"int: Median = {query3}");

string[] numbers4 = ["one", "two", "three", "four", "five"];

// With the generic overload, you can also use numeric properties of objects.
var query4 = numbers4.Median(str => str.Length);

Console.WriteLine($"string: Median = {query4}");
// This code produces the following output:
//     int: Median = 3
//     string: Median = 4
```

Extend the IEnumerable<T> interface with a custom query method that returns a *sequence of values*. In this case, the method must return a collection of type IEnumerable<T>. Such

methods can be used to apply filters or data transforms to a sequence of values.

The following example shows how to create an extension method named `AlternateElements` that returns every other element in a collection, starting from the first element.

C#

```
// Extension method for the IEnumerable<T> interface.
// The method returns every other element of a sequence.
public static IEnumerable<T> AlternateElements<T>(this IEnumerable<T> source)
{
    int index = 0;
    foreach (T element in source)
    {
        if (index % 2 == 0)
        {
            yield return element;
        }

        index++;
    }
}
```

Call this extension method for any enumerable collection just as you would call other methods from the IEnumerable<T> interface, as shown in the following code:

C#

```
string[] strings = ["a", "b", "c", "d", "e"];

var query5 = strings.AlternateElements();

foreach (var element in query5)
{
    Console.WriteLine(element);
}
// This code produces the following output:
//     a
//     c
//     e
```

Each example shown in this article has a different *receiver*. That means you must declare each method in a different extension block that specifies the unique receiver. The following code example shows a single static class with three different extension blocks, each of which contains one of the methods defined in this article:

C#

```
public static class EnumerableExtension
{
```

```
    extension(IEnumerable<double>? source)
    {
        public double Median()
        {
            if (source is null || !source.Any())
            {
                throw new InvalidOperationException("Cannot compute median for a
null or empty set.");
            }

            var sortedList =
                source.OrderBy(number => number).ToList();

            int itemIndex = sortedList.Count / 2;

            if (sortedList.Count % 2 == 0)
            {
                // Even number of items.
                return (sortedList[itemIndex] + sortedList[itemIndex - 1]) / 2;
            }
            else
            {
                // Odd number of items.
                return sortedList[itemIndex];
            }
        }
    }

    extension(IEnumerable<int> source)
    {
        public double Median() =>
            (from number in source select (double)number).Median();
    }

    extension<T>(IEnumerable<T> source)
    {
        public double Median(Func<T, double> selector) =>
            (from num in source select selector(num)).Median();

        public IEnumerable<T> AlternateElements()
        {
            int index = 0;
            foreach (T element in source)
            {
                if (index % 2 == 0)
                {
                    yield return element;
                }

                index++;
            }
        }
    }
}
```

The final extension block declares a generic extension block. The type parameter for the receiver is declared on the `extension` itself.

The preceding example declares one extension member in each extension block. In most cases, you create multiple extension members for the same receiver. In those cases, declare the extensions for those members in a single extension block.

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 12/03/2025

Query based on run-time state

Article • 04/25/2024

In most LINQ queries, the general shape of the query is set in code. You might filter items using a `where` clause, sort the output collection using `orderby`, group items, or perform some computation. Your code might provide parameters for the filter, or the sort key, or other expressions that are part of the query. However, the overall shape of the query can't change. In this article, you learn techniques to use System.Linq.IQueryable<T> interface and types that implement it to modify the shape of a query at run time.

You use these techniques to build queries at run time, where some user input or run-time state changes the query methods you want to use as part of the query. You want to edit the query by adding, removing, or modifying query clauses.

> **Note**
>
> Make sure you add `using System.Linq.Expressions;` and `using static System.Linq.Expressions.Expression;` at the top of your *.cs* file.

Consider code that defines an IQueryable or an IQueryable<T> against a data source:

C#

```
string[] companyNames = [
    "Consolidated Messenger", "Alpine Ski House", "Southridge Video",
    "City Power & Light", "Coho Winery", "Wide World Importers",
    "Graphic Design Institute", "Adventure Works", "Humongous Insurance",
    "Woodgrove Bank", "Margie's Travel", "Northwind Traders",
    "Blue Yonder Airlines", "Trey Research", "The Phone Company",
    "Wingtip Toys", "Lucerne Publishing", "Fourth Coffee"
];

// Use an in-memory array as the data source, but the IQueryable could have
come
// from anywhere -- an ORM backed by a database, a web request, or any other
LINQ provider.
IQueryable<string> companyNamesSource = companyNames.AsQueryable();
var fixedQry = companyNames.OrderBy(x => x);
```

Every time you run the preceding code, the same exact query is executed. Let's learn how to modify the query extend it or modify it. Fundamentally, an IQueryable has two components:

- Expression—a language-agnostic and datasource-agnostic representation of the current query's components, in the form of an expression tree.
- Provider—an instance of a LINQ provider, which knows how to materialize the current query into a value or set of values.

In the context of dynamic querying, the provider usually remains the same; the expression tree of the query differs from query to query.

Expression trees are immutable; if you want a different expression tree—and thus a different query—you need to translate the existing expression tree to a new one. The following sections describe specific techniques for querying differently in response to run-time state:

- Use run-time state from within the expression tree
- Call more LINQ methods
- Vary the expression tree passed into the LINQ methods
- Construct an Expression<TDelegate> expression tree using the factory methods at Expression
- Add method call nodes to an IQueryable's expression tree
- Construct strings, and use the Dynamic LINQ library

Each of techniques enables more capabilities, but at a cost of increased complexity.

Use run-time state from within the expression tree

The simplest way to query dynamically is to reference the run-time state directly in the query via a closed-over variable, such as `length` in the following code example:

C#

```
var length = 1;
var qry = companyNamesSource
    .Select(x => x.Substring(0, length))
    .Distinct();

Console.WriteLine(string.Join(",", qry));
// prints: C, A, S, W, G, H, M, N, B, T, L, F

length = 2;
Console.WriteLine(string.Join(",", qry));
// prints: Co, Al, So, Ci, Wi, Gr, Ad, Hu, Wo, Ma, No, Bl, Tr, Th, Lu, Fo
```

The internal expression tree—and thus the query—isn't modified; the query returns different values only because the value of `length` changed.

Call more LINQ methods

Generally, the built-in LINQ methods at Queryable perform two steps:

- Wrap the current expression tree in a MethodCallExpression representing the method call.
- Pass the wrapped expression tree back to the provider, either to return a value via the provider's IQueryProvider.Execute method; or to return a translated query object via the IQueryProvider.CreateQuery method.

You can replace the original query with the result of an System.Linq.IQueryable<T>-returning method, to get a new query. You can use run-time state, as in the following example:

C#

```
// bool sortByLength = /* ... */;

var qry = companyNamesSource;
if (sortByLength)
{
    qry = qry.OrderBy(x => x.Length);
}
```

Vary the expression tree passed into the LINQ methods

You can pass in different expressions to the LINQ methods, depending on run-time state:

C#

```
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>> expr = (startsWith, endsWith) switch
{
    ("" or null, "" or null) => x => true,
    (_, "" or null) => x => x.StartsWith(startsWith),
    ("" or null, _) => x => x.EndsWith(endsWith),
    (_, _) => x => x.StartsWith(startsWith) || x.EndsWith(endsWith)
};
```

```
var qry = companyNamesSource.Where(expr);
```

You might also want to compose the various subexpressions using another library such as LinqKit's PredicateBuilder:

C#

```
// This is functionally equivalent to the previous example.

// using LinqKit;
// string? startsWith = /* ... */;
// string? endsWith = /* ... */;

Expression<Func<string, bool>>? expr = PredicateBuilder.New<string>(false);
var original = expr;
if (!string.IsNullOrEmpty(startsWith))
{
    expr = expr.Or(x => x.StartsWith(startsWith));
}
if (!string.IsNullOrEmpty(endsWith))
{
    expr = expr.Or(x => x.EndsWith(endsWith));
}
if (expr == original)
{
    expr = x => true;
}

var qry = companyNamesSource.Where(expr);
```

Construct expression trees and queries using factory methods

In all the examples up to this point, you know the element type at compile time—`string`—and thus the type of the query—`IQueryable<string>`. You might add components to a query of any element type, or to add different components, depending on the element type. You can create expression trees from the ground up, using the factory methods at System.Linq.Expressions.Expression, and thus tailor the expression at run time to a specific element type.

Constructing an Expression<TDelegate>

When you construct an expression to pass into one of the LINQ methods, you're actually constructing an instance of System.Linq.Expressions.Expression<TDelegate>, where

`TDelegate` is some delegate type such as `Func<string, bool>`, `Action`, or a custom delegate type.

System.Linq.Expressions.Expression<TDelegate> inherits from LambdaExpression, which represents a complete lambda expression like the following example:

```
C#

Expression<Func<string, bool>> expr = x => x.StartsWith("a");
```

A LambdaExpression has two components:

1. A parameter list—`(string x)`—represented by the Parameters property.
2. A body—`x.StartsWith("a")`—represented by the Body property.

The basic steps in constructing an Expression<TDelegate> are as follows:

1. Define ParameterExpression objects for each of the parameters (if any) in the lambda expression, using the Parameter factory method.

   ```
   C#

   ParameterExpression x = Parameter(typeof(string), "x");
   ```

2. Construct the body of your LambdaExpression, using the ParameterExpression defined, and the factory methods at Expression. For instance, an expression representing `x.StartsWith("a")` could be constructed like this:

   ```
   C#

   Expression body = Call(
       x,
       typeof(string).GetMethod("StartsWith", [typeof(string)])!,
       Constant("a")
   );
   ```

3. Wrap the parameters and body in a compile-time-typed Expression<TDelegate>, using the appropriate Lambda factory method overload:

   ```
   C#

   Expression<Func<string, bool>> expr = Lambda<Func<string, bool>>(body,
   x);
   ```

The following sections describe a scenario in which you might want to construct an Expression<TDelegate> to pass into a LINQ method. It provides a complete example of how to do so using the factory methods.

Construct a full query at run time

You want to write queries that work with multiple entity types:

C#

```
record Person(string LastName, string FirstName, DateTime DateOfBirth);
record Car(string Model, int Year);
```

For any of these entity types, you want to filter and return only those entities that have a given text inside one of their `string` fields. For `Person`, you'd want to search the `FirstName` and `LastName` properties:

C#

```
string term = /* ... */;
var personsQry = new List<Person>()
    .AsQueryable()
    .Where(x => x.FirstName.Contains(term) || x.LastName.Contains(term));
```

But for `Car`, you'd want to search only the `Model` property:

C#

```
string term = /* ... */;
var carsQry = new List<Car>()
    .AsQueryable()
    .Where(x => x.Model.Contains(term));
```

While you could write one custom function for `IQueryable<Person>` and another for `IQueryable<Car>`, the following function adds this filtering to any existing query, irrespective of the specific element type.

C#

```
// using static System.Linq.Expressions.Expression;

IQueryable<T> TextFilter<T>(IQueryable<T> source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
```

```
    // T is a compile-time placeholder for the element type of the query.
    Type elementType = typeof(T);

    // Get all the string properties on this specific type.
    PropertyInfo[] stringProperties = elementType
        .GetProperties()
        .Where(x => x.PropertyType == typeof(string))
        .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Get the right overload of String.Contains
    MethodInfo containsMethod = typeof(string).GetMethod("Contains",
[typeof(string)])!;

    // Create a parameter for the expression tree:
    // the 'x' in 'x => x.PropertyName.Contains("term")'
    // The type of this parameter is the query's element type
    ParameterExpression prm = Parameter(elementType);

    // Map each property to an expression tree node
    IEnumerable<Expression> expressions = stringProperties
        .Select(prp =>
            // For each property, we have to construct an expression tree
node like x.PropertyName.Contains("term")
            Call(                   // .Contains(...)
                Property(           // .PropertyName
                    prm,            // x
                    prp
                ),
                containsMethod,
                Constant(term)      // "term"
            )
        );

    // Combine all the resultant expression nodes using ||
    Expression body = expressions
        .Aggregate((prev, current) => Or(prev, current));

    // Wrap the expression body in a compile-time-typed lambda expression
    Expression<Func<T, bool>> lambda = Lambda<Func<T, bool>>(body, prm);

    // Because the lambda is compile-time-typed (albeit with a generic
parameter), we can use it with the Where method
    return source.Where(lambda);
}
```

Because the `TextFilter` function takes and returns an IQueryable<T> (and not just an IQueryable), you can add further compile-time-typed query elements after the text filter.

C#

```
var qry = TextFilter(
        new List<Person>().AsQueryable(),
```

```
        "abcd"
    )
    .Where(x => x.DateOfBirth < new DateTime(2001, 1, 1));

var qry1 = TextFilter(
        new List<Car>().AsQueryable(),
        "abcd"
    )
    .Where(x => x.Year == 2010);
```

Add method call nodes to the IQueryable<TDelegate>'s expression tree

If you have an IQueryable instead of an IQueryable<T>, you can't directly call the generic LINQ methods. One alternative is to build the inner expression tree as shown in the previous example, and use reflection to invoke the appropriate LINQ method while passing in the expression tree.

You could also duplicate the LINQ method's functionality, by wrapping the entire tree in a MethodCallExpression that represents a call to the LINQ method:

C#

```
IQueryable TextFilter_Untyped(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }
    Type elementType = source.ElementType;

    // The logic for building the ParameterExpression and the
LambdaExpression's body is the same as in the previous example,
    // but has been refactored into the constructBody function.
    (Expression? body, ParameterExpression? prm) =
constructBody(elementType, term);
    if (body is null) { return source; }

    Expression filteredTree = Call(
        typeof(Queryable),
        "Where",
        [elementType],
        source.Expression,
        Lambda(body, prm!)
    );

    return source.Provider.CreateQuery(filteredTree);
}
```

In this case, you don't have a compile-time `T` generic placeholder, so you use the Lambda overload that doesn't require compile-time type information, and which

produces a LambdaExpression instead of an Expression<TDelegate>.

The Dynamic LINQ library

Constructing expression trees using factory methods is relatively complex; it's easier to compose strings. The Dynamic LINQ library exposes a set of extension methods on IQueryable corresponding to the standard LINQ methods at Queryable, and which accept strings in a special syntax instead of expression trees. The library generates the appropriate expression tree from the string, and can return the resultant translated IQueryable.

For instance, the previous example could be rewritten as follows:

C#

```
// using System.Linq.Dynamic.Core

IQueryable TextFilter_Strings(IQueryable source, string term)
{
    if (string.IsNullOrEmpty(term)) { return source; }

    var elementType = source.ElementType;

    // Get all the string property names on this specific type.
    var stringProperties =
        elementType.GetProperties()
            .Where(x => x.PropertyType == typeof(string))
            .ToArray();
    if (!stringProperties.Any()) { return source; }

    // Build the string expression
    string filterExpr = string.Join(
        " || ",
        stringProperties.Select(prp => $"{prp.Name}.Contains(@0)")
    );

    return source.Where(filterExpr, term);
}
```

Asynchronous programming with async and await

07/16/2025

The Task asynchronous programming (TAP) model provides a layer of abstraction over typical asynchronous coding. In this model, you write code as a sequence of statements, the same as usual. The difference is you can read your task-based code as the compiler processes each statement and before it starts processing the next statement. To accomplish this model, the compiler performs many transformations to complete each task. Some statements can initiate work and return a Task object that represents the ongoing work and the compiler must resolve these transformations. The goal of task asynchronous programming is to enable code that reads like a sequence of statements, but executes in a more complicated order. Execution is based on external resource allocation and when tasks complete.

The task asynchronous programming model is analogous to how people give instructions for processes that include asynchronous tasks. This article uses an example with instructions for making breakfast to show how the `async` and `await` keywords make it easier to reason about code that includes a series of asynchronous instructions. The instructions for making a breakfast might be provided as a list:

1. Pour a cup of coffee.
2. Heat a pan, then fry two eggs.
3. Cook three hash brown patties.
4. Toast two pieces of bread.
5. Spread butter and jam on the toast.
6. Pour a glass of orange juice.

If you have experience with cooking, you might complete these instructions **asynchronously**. You start warming the pan for eggs, then start cooking the hash browns. You put the bread in the toaster, then start cooking the eggs. At each step of the process, you start a task, and then transition to other tasks that are ready for your attention.

Cooking breakfast is a good example of asynchronous work that isn't parallel. One person (or thread) can handle all the tasks. One person can make breakfast asynchronously by starting the next task before the previous task completes. Each cooking task progresses regardless of whether someone is actively watching the process. As soon as you start warming the pan for the eggs, you can begin cooking the hash browns. After the hash browns start to cook, you can put the bread in the toaster.

For a parallel algorithm, you need multiple people who cook (or multiple threads). One person cooks the eggs, another cooks the hash browns, and so on. Each person focuses on their one

specific task. Each person who is cooking (or each thread) is blocked synchronously waiting for the current task to complete: Hash browns ready to flip, bread ready to pop up in toaster, and so on.



Consider the same list of synchronous instructions written as C# code statements:

C#

```
using System;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this example. They
are simply marker classes for the purpose of demonstration, contain no properties,
and serve no other purpose.
    internal class HashBrown { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
```

```
        static void Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            Egg eggs = FryEggs(2);
            Console.WriteLine("eggs are ready");

            HashBrown hashBrown = FryHashBrowns(3);
            Console.WriteLine("hash browns are ready");

            Toast toast = ToastBread(2);
            ApplyButter(toast);
            ApplyJam(toast);
            Console.WriteLine("toast is ready");

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");

        private static Toast ToastBread(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the toaster");
            }
            Console.WriteLine("Start toasting...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static HashBrown FryHashBrowns(int patties)
        {
            Console.WriteLine($"putting {patties} hash brown patties in the pan");
            Console.WriteLine("cooking first side of hash browns...");
            Task.Delay(3000).Wait();
            for (int patty = 0; patty < patties; patty++)
            {
                Console.WriteLine("flipping a hash brown patty");
            }
```

```
            Console.WriteLine("cooking the second side of hash browns...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put hash browns on plate");

            return new HashBrown();
        }

        private static Egg FryEggs(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            Task.Delay(3000).Wait();
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            Task.Delay(3000).Wait();
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

If you interpret these instructions as a computer would, breakfast takes about 30 minutes to prepare. The duration is the sum of the individual task times. The computer blocks for each statement until all work completes, and then it proceeds to the next task statement. This approach can take significant time. In the breakfast example, the computer method creates an unsatisfying breakfast. Later tasks in the synchronous list, like toasting the bread, don't start until earlier tasks complete. Some food gets cold before the breakfast is ready to serve.

If you want the computer to execute instructions asynchronously, you must write asynchronous code. When you write client programs, you want the UI to be responsive to user input. Your application shouldn't freeze all interaction while downloading data from the web. When you write server programs, you don't want to block threads that might be serving other requests. Using synchronous code when asynchronous alternatives exist hurts your ability to scale out less expensively. You pay for blocked threads.

Successful modern apps require asynchronous code. Without language support, writing asynchronous code requires callbacks, completion events, or other means that obscure the original intent of the code. The advantage of synchronous code is the step-by-step action that makes it easy to scan and understand. Traditional asynchronous models force you to focus on the asynchronous nature of the code, not on the fundamental actions of the code.

Don't block, await instead

The previous code highlights an unfortunate programming practice: Writing synchronous code to perform asynchronous operations. The code blocks the current thread from doing any other work. The code doesn't interrupt the thread while there are running tasks. The outcome of this model is similar to staring at the toaster after you put in the bread. You ignore any interruptions and don't start other tasks until the bread pops up. You don't take the butter and jam out of the fridge. You might miss seeing a fire starting on the stove. You want to both toast the bread and handle other concerns at the same time. The same is true with your code.

You can start by updating the code so the thread doesn't block while tasks are running. The `await` keyword provides a nonblocking way to start a task, then continue execution when the task completes. A simple asynchronous version of the breakfast code looks like the following snippet:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    Egg eggs = await FryEggsAsync(2);
    Console.WriteLine("eggs are ready");

    HashBrown hashBrown = await FryHashBrownsAsync(3);
    Console.WriteLine("hash browns are ready");

    Toast toast = await ToastBreadAsync(2);
    ApplyButter(toast);
    ApplyJam(toast);
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

The code updates the original method bodies of `FryEggs`, `FryHashBrowns`, and `ToastBread` to return `Task<Egg>`, `Task<HashBrown>`, and `Task<Toast>` objects, respectively. The updated method names include the "Async" suffix: `FryEggsAsync`, `FryHashBrownsAsync`, and `ToastBreadAsync`. The `Main` method returns the `Task` object, although it doesn't have a `return` expression, which is by design. For more information, see Evaluation of a void-returning async function.

> **Note**
>
> The updated code doesn't yet take advantage of key features of asynchronous programming, which can result in shorter completion times. The code processes the tasks in roughly the same amount of time as the initial synchronous version. For the full method implementations, see the final version of the code later in this article.

Let's apply the breakfast example to the updated code. The thread doesn't block while the eggs or hash browns are cooking, but the code also doesn't start other tasks until the current work completes. You still put the bread in the toaster and stare at the toaster until the bread pops up, but you can now respond to interruptions. In a restaurant where multiple orders are placed, the cook can start a new order while another is already cooking.

In the updated code, the thread working on the breakfast isn't blocked while waiting for any started task that's unfinished. For some applications, this change is all you need. You can enable your app to support user interaction while data downloads from the web. In other scenarios, you might want to start other tasks while waiting for the previous task to complete.

Start tasks concurrently

For most operations, you want to start several independent tasks immediately. As each task completes, you initiate other work that's ready to start. When you apply this methodology to the breakfast example, you can prepare breakfast more quickly. You also get everything ready close to the same time, so you can enjoy a hot breakfast.

The System.Threading.Tasks.Task class and related types are classes you can use to apply this style of reasoning to tasks that are in progress. This approach enables you to write code that more closely resembles the way you create breakfast in real life. You start cooking the eggs, hash browns, and toast at the same time. As each food item requires action, you turn your attention to that task, take care of the action, and then wait for something else that requires your attention.

In your code, you start a task and hold on to the Task object that represents the work. You use the `await` method on the task to delay acting on the work until the result is ready.

Apply these changes to the breakfast code. The first step is to store the tasks for operations when they start, rather than using the `await` expression:

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");
```

```
Task<Egg> eggsTask = FryEggsAsync(2);
Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");

Task<HashBrown> hashBrownTask = FryHashBrownsAsync(3);
HashBrown hashBrown = await hashBrownTask;
Console.WriteLine("Hash browns are ready");

Task<Toast> toastTask = ToastBreadAsync(2);
Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");

Juice oj = PourOJ();
Console.WriteLine("Oj is ready");
Console.WriteLine("Breakfast is ready!");
```

These revisions don't help to get your breakfast ready any faster. The `await` expression is applied to all tasks as soon as they start. The next step is to move the `await` expressions for the hash browns and eggs to the end of the method, before you serve the breakfast:

C#

```
Coffee cup = PourCoffee();
Console.WriteLine("Coffee is ready");

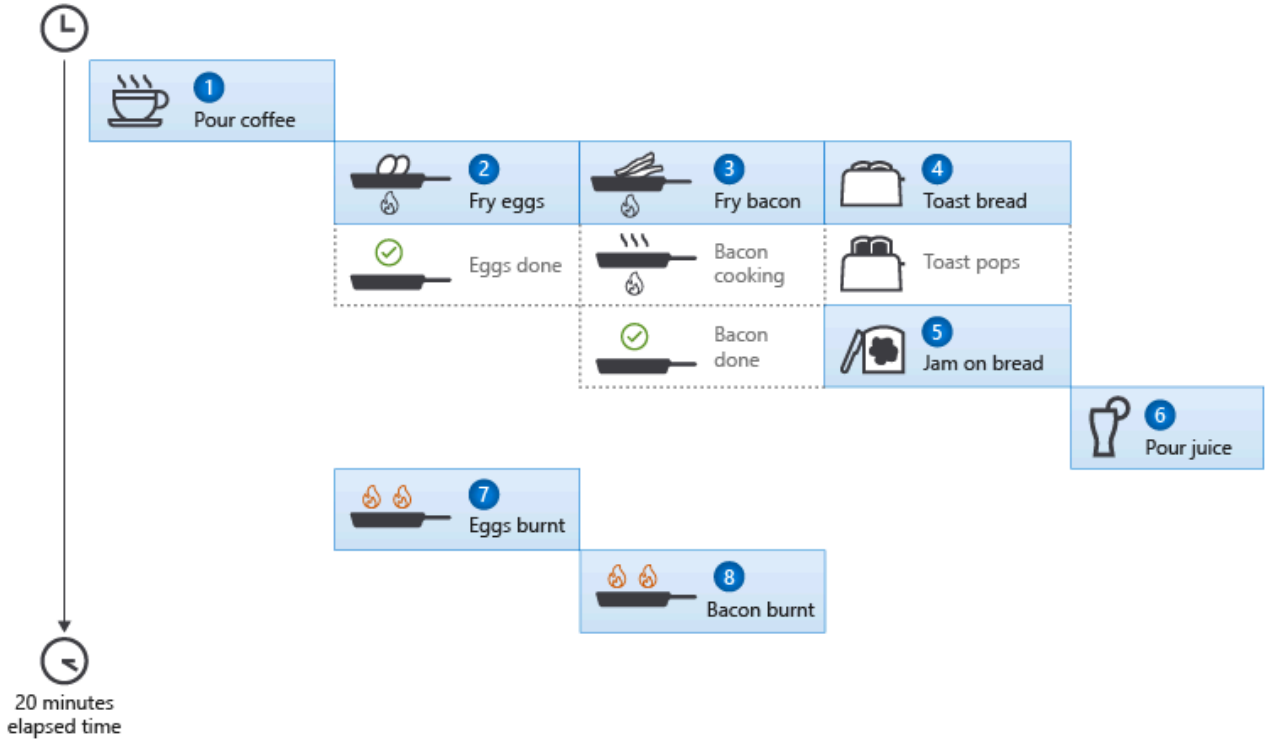
Task<Egg> eggsTask = FryEggsAsync(2);
Task<HashBrown> hashBrownTask = FryHashBrownsAsync(3);
Task<Toast> toastTask = ToastBreadAsync(2);

Toast toast = await toastTask;
ApplyButter(toast);
ApplyJam(toast);
Console.WriteLine("Toast is ready");
Juice oj = PourOJ();
Console.WriteLine("Oj is ready");

Egg eggs = await eggsTask;
Console.WriteLine("Eggs are ready");
HashBrown hashBrown = await hashBrownTask;
Console.WriteLine("Hash browns are ready");

Console.WriteLine("Breakfast is ready!");
```

You now have an asynchronously prepared breakfast that takes about 20 minutes to prepare. The total cook time is reduced because some tasks run concurrently.



The code updates improve the preparation process by reducing the cook time, but they introduce a regression by burning the eggs and hash browns. You start all the asynchronous tasks at once. You wait on each task only when you need the results. The code might be similar to program in a web application that makes requests to different microservices and then combines the results into a single page. You make all the requests immediately, and then apply the `await` expression on all those tasks and compose the web page.

Support composition with tasks

The previous code revisions help get everything ready for breakfast at the same time, except the toast. The process of making the toast is a *composition* of an asynchronous operation (toast the bread) with synchronous operations (spread butter and jam on the toast). This example illustrates an important concept about asynchronous programming:

> **ⓘ Important**
>
> The composition of an asynchronous operation followed by synchronous work is an asynchronous operation. Stated another way, if any portion of an operation is asynchronous, the entire operation is asynchronous.

In the previous updates, you learned how to use Task or Task<TResult> objects to hold running tasks. You wait on each task before you use its result. The next step is to create methods that represent the combination of other work. Before you serve breakfast, you want to wait on the task that represents toasting the bread before you spread the butter and jam.

You can represent this work with the following code:

C#

```
static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
{
    var toast = await ToastBreadAsync(number);
    ApplyButter(toast);
    ApplyJam(toast);

    return toast;
}
```

The `MakeToastWithButterAndJamAsync` method has the `async` modifier in its signature that signals to the compiler that the method contains an `await` expression and contains asynchronous operations. The method represents the task that toasts the bread, then spreads the butter and jam. The method returns a Task<TResult> object that represents the composition of the three operations.

The revised main block of code now looks like this:

C#

```
static async Task Main(string[] args)
{
    Coffee cup = PourCoffee();
    Console.WriteLine("coffee is ready");

    var eggsTask = FryEggsAsync(2);
    var hashBrownTask = FryHashBrownsAsync(3);
    var toastTask = MakeToastWithButterAndJamAsync(2);

    var eggs = await eggsTask;
    Console.WriteLine("eggs are ready");

    var hashBrown = await hashBrownTask;
    Console.WriteLine("hash browns are ready");

    var toast = await toastTask;
    Console.WriteLine("toast is ready");

    Juice oj = PourOJ();
    Console.WriteLine("oj is ready");
    Console.WriteLine("Breakfast is ready!");
}
```

This code change illustrates an important technique for working with asynchronous code. You compose tasks by separating the operations into a new method that returns a task. You can choose when to wait on that task. You can start other tasks concurrently.

Handle asynchronous exceptions

Up to this point, your code implicitly assumes all tasks complete successfully. Asynchronous methods throw exceptions, just like their synchronous counterparts. The goals for asynchronous support for exceptions and error handling are the same as for asynchronous support in general. The best practice is to write code that reads like a series of synchronous statements. Tasks throw exceptions when they can't complete successfully. The client code can catch those exceptions when the `await` expression is applied to a started task.

In the breakfast example, suppose the toaster catches fire while toasting the bread. You can simulate that problem by modifying the `ToastBreadAsync` method to match the following code:

C#

```
private static async Task<Toast> ToastBreadAsync(int slices)
{
    for (int slice = 0; slice < slices; slice++)
    {
        Console.WriteLine("Putting a slice of bread in the toaster");
    }
    Console.WriteLine("Start toasting...");
    await Task.Delay(2000);
    Console.WriteLine("Fire! Toast is ruined!");
    throw new InvalidOperationException("The toaster is on fire");
    await Task.Delay(1000);
    Console.WriteLine("Remove toast from toaster");

    return new Toast();
}
```

> **Note**
>
> When you compile this code, you see a warning about unreachable code. This error is by design. After the toaster catches fire, operations don't proceed normally and the code returns an error.

After you make the code changes, run the application and check the output:

Console

```
Pouring coffee
Coffee is ready
Warming the egg pan...
putting 3 hash brown patties in the pan
Cooking first side of hash browns...
Putting a slice of bread in the toaster
```

```
Putting a slice of bread in the toaster
Start toasting...
Fire! Toast is ruined!
Flipping a hash brown patty
Flipping a hash brown patty
Flipping a hash brown patty
Cooking the second side of hash browns...
Cracking 2 eggs
Cooking the eggs ...
Put hash browns on plate
Put eggs on plate
Eggs are ready
Hash browns are ready
Unhandled exception. System.InvalidOperationException: The toaster is on fire
   at AsyncBreakfast.Program.ToastBreadAsync(Int32 slices) in Program.cs:line 65
   at AsyncBreakfast.Program.MakeToastWithButterAndJamAsync(Int32 number) in
Program.cs:line 36
   at AsyncBreakfast.Program.Main(String[] args) in Program.cs:line 24
   at AsyncBreakfast.Program.<Main>(String[] args)
```

Notice that quite a few tasks finish between the time when the toaster catches fire and the system observes the exception. When a task that runs asynchronously throws an exception, that task is **faulted**. The `Task` object holds the exception thrown in the Task.Exception property. Faulted tasks throw an exception when the `await` expression is applied to the task.

There are two important mechanisms to understand about this process:

- How an exception is stored in a faulted task
- How an exception is unpackaged and rethrown when code waits (`await`) on a faulted task

When code running asynchronously throws an exception, the exception is stored in the `Task` object. The Task.Exception property is a System.AggregateException object because more than one exception might be thrown during asynchronous work. Any exception thrown is added to the AggregateException.InnerExceptions collection. If the `Exception` property is null, a new `AggregateException` object is created and the thrown exception is the first item in the collection.

The most common scenario for a faulted task is that the `Exception` property contains exactly one exception. When your code waits on a faulted task, it rethrows the first AggregateException.InnerExceptions exception in the collection. This result is the reason why the output from the example shows an System.InvalidOperationException object rather than an `AggregateException` object. Extracting the first inner exception makes working with asynchronous methods as similar as possible to working with their synchronous counterparts. You can examine the `Exception` property in your code when your scenario might generate multiple exceptions.

> **Tip**
>
> The recommended practice is for any argument validation exceptions to emerge *synchronously* from task-returning methods. For more information and examples, see Exceptions in task-returning methods.

Before you continue to the next section, comment out the following two statements in your `ToastBreadAsync` method. You don't want to start another fire:

```
C#

Console.WriteLine("Fire! Toast is ruined!");
throw new InvalidOperationException("The toaster is on fire");
```

Apply await expressions to tasks efficiently

You can improve the series of `await` expressions at the end of the previous code by using methods of the `Task` class. One API is the WhenAll method, which returns a Task object that completes when all the tasks in its argument list are complete. The following code demonstrates this method:

```
C#

await Task.WhenAll(eggsTask, hashBrownTask, toastTask);
Console.WriteLine("Eggs are ready");
Console.WriteLine("Hash browns are ready");
Console.WriteLine("Toast is ready");
Console.WriteLine("Breakfast is ready!");
```

Another option is to use the WhenAny method, which returns a `Task<Task>` object that completes when any of its arguments complete. You can wait on the returned task because you know the task is done. The following code shows how you can use the WhenAny method to wait on the first task to finish and then process its result. After you process the result from the completed task, you remove the completed task from the list of tasks passed to the `WhenAny` method.

```
C#

var breakfastTasks = new List<Task> { eggsTask, hashBrownTask, toastTask };
while (breakfastTasks.Count > 0)
{
    Task finishedTask = await Task.WhenAny(breakfastTasks);
    if (finishedTask == eggsTask)
```

```
    {
        Console.WriteLine("Eggs are ready");
    }
    else if (finishedTask == hashBrownTask)
    {
        Console.WriteLine("Hash browns are ready");
    }
    else if (finishedTask == toastTask)
    {
        Console.WriteLine("Toast is ready");
    }
    await finishedTask;
    breakfastTasks.Remove(finishedTask);
}
```

Near the end of the code snippet, notice the `await finishedTask;` expression. The `await Task.WhenAny` expression doesn't wait on the finished task, but rather waits on the `Task` object returned by the `Task.WhenAny` method. The result of the `Task.WhenAny` method is the completed (or faulted) task. The best practice is to wait on the task again, even when you know the task is complete. In this manner, you can retrieve the task result, or ensure any exception that causes the task to fault is thrown.

Review final code

Here's what the final version of the code looks like:

C#

```
using System;
using System.Collections.Generic;
using System.Threading.Tasks;

namespace AsyncBreakfast
{
    // These classes are intentionally empty for the purpose of this example. They
are simply marker classes for the purpose of demonstration, contain no properties,
and serve no other purpose.
    internal class HashBrown { }
    internal class Coffee { }
    internal class Egg { }
    internal class Juice { }
    internal class Toast { }

    class Program
    {
```

```
        static async Task Main(string[] args)
        {
            Coffee cup = PourCoffee();
            Console.WriteLine("coffee is ready");

            var eggsTask = FryEggsAsync(2);
            var hashBrownTask = FryHashBrownsAsync(3);
            var toastTask = MakeToastWithButterAndJamAsync(2);

            var breakfastTasks = new List<Task> { eggsTask, hashBrownTask,
toastTask };
            while (breakfastTasks.Count > 0)
            {
                Task finishedTask = await Task.WhenAny(breakfastTasks);
                if (finishedTask == eggsTask)
                {
                    Console.WriteLine("eggs are ready");
                }
                else if (finishedTask == hashBrownTask)
                {
                    Console.WriteLine("hash browns are ready");
                }
                else if (finishedTask == toastTask)
                {
                    Console.WriteLine("toast is ready");
                }
                await finishedTask;
                breakfastTasks.Remove(finishedTask);
            }

            Juice oj = PourOJ();
            Console.WriteLine("oj is ready");
            Console.WriteLine("Breakfast is ready!");
        }

        static async Task<Toast> MakeToastWithButterAndJamAsync(int number)
        {
            var toast = await ToastBreadAsync(number);
            ApplyButter(toast);
            ApplyJam(toast);

            return toast;
        }

        private static Juice PourOJ()
        {
            Console.WriteLine("Pouring orange juice");
            return new Juice();
        }

        private static void ApplyJam(Toast toast) =>
            Console.WriteLine("Putting jam on the toast");

        private static void ApplyButter(Toast toast) =>
            Console.WriteLine("Putting butter on the toast");
```

```
        private static async Task<Toast> ToastBreadAsync(int slices)
        {
            for (int slice = 0; slice < slices; slice++)
            {
                Console.WriteLine("Putting a slice of bread in the toaster");
            }
            Console.WriteLine("Start toasting...");
            await Task.Delay(3000);
            Console.WriteLine("Remove toast from toaster");

            return new Toast();
        }

        private static async Task<HashBrown> FryHashBrownsAsync(int patties)
        {
            Console.WriteLine($"putting {patties} hash brown patties in the pan");
            Console.WriteLine("cooking first side of hash browns...");
            await Task.Delay(3000);
            for (int patty = 0; patty < patties; patty++)
            {
                Console.WriteLine("flipping a hash brown patty");
            }
            Console.WriteLine("cooking the second side of hash browns...");
            await Task.Delay(3000);
            Console.WriteLine("Put hash browns on plate");

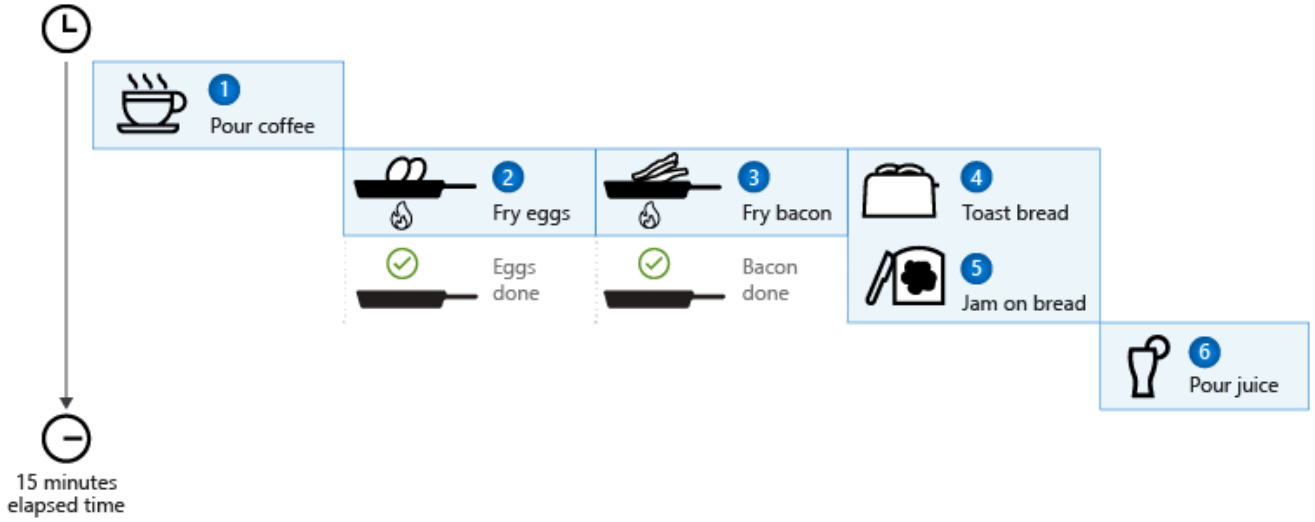
            return new HashBrown();
        }

        private static async Task<Egg> FryEggsAsync(int howMany)
        {
            Console.WriteLine("Warming the egg pan...");
            await Task.Delay(3000);
            Console.WriteLine($"cracking {howMany} eggs");
            Console.WriteLine("cooking the eggs ...");
            await Task.Delay(3000);
            Console.WriteLine("Put eggs on plate");

            return new Egg();
        }

        private static Coffee PourCoffee()
        {
            Console.WriteLine("Pouring coffee");
            return new Coffee();
        }
    }
}
```

The code completes the asynchronous breakfast tasks in about 15 minutes. The total time is reduced because some tasks run concurrently. The code simultaneously monitors multiple tasks and takes action only as needed.



The final code is asynchronous. It more accurately reflects how a person might cook breakfast. Compare the final code with the first code sample in the article. The core actions are still clear by reading the code. You can read the final code the same way you read the list of instructions for making a breakfast, as shown at the beginning of the article. The language features for the `async` and `await` keywords provide the translation every person makes to follow the written instructions: Start tasks as you can and don't block while waiting for tasks to complete.

Async/await vs ContinueWith

The `async` and `await` keywords provide syntactic simplification over using Task.ContinueWith directly. While `async`/`await` and `ContinueWith` have similar semantics for handling asynchronous operations, the compiler doesn't necessarily translate `await` expressions directly into `ContinueWith` method calls. Instead, the compiler generates optimized state machine code that provides the same logical behavior. This transformation provides significant readability and maintainability benefits, especially when chaining multiple asynchronous operations.

Consider a scenario where you need to perform multiple sequential asynchronous operations. Here's how the same logic looks when implemented with `ContinueWith` compared to `async`/`await`:

Using ContinueWith

With `ContinueWith`, each step in a sequence of asynchronous operations requires nested continuations:

C#

```
// Using ContinueWith - demonstrates the complexity when chaining operations
static Task MakeBreakfastWithContinueWith()
```

```
{
    return StartCookingEggsAsync()
        .ContinueWith(eggsTask =>
        {
            var eggs = eggsTask.Result;
            Console.WriteLine("Eggs ready, starting bacon...");
            return StartCookingBaconAsync();
        })
        .Unwrap()
        .ContinueWith(baconTask =>
        {
            var bacon = baconTask.Result;
            Console.WriteLine("Bacon ready, starting toast...");
            return StartToastingBreadAsync();
        })
        .Unwrap()
        .ContinueWith(toastTask =>
        {
            var toast = toastTask.Result;
            Console.WriteLine("Toast ready, applying butter...");
            return ApplyButterAsync(toast);
        })
        .Unwrap()
        .ContinueWith(butteredToastTask =>
        {
            var butteredToast = butteredToastTask.Result;
            Console.WriteLine("Butter applied, applying jam...");
            return ApplyJamAsync(butteredToast);
        })
        .Unwrap()
        .ContinueWith(finalToastTask =>
        {
            var finalToast = finalToastTask.Result;
            Console.WriteLine("Breakfast completed with ContinueWith!");
        });
}
```

Using async/await

The same sequence of operations using `async`/`await` reads much more naturally:

C#

```
// Using async/await - much cleaner and easier to read
static async Task MakeBreakfastWithAsyncAwait()
{
    var eggs = await StartCookingEggsAsync();
    Console.WriteLine("Eggs ready, starting bacon...");

    var bacon = await StartCookingBaconAsync();
    Console.WriteLine("Bacon ready, starting toast...");
```

```
    var toast = await StartToastingBreadAsync();
    Console.WriteLine("Toast ready, applying butter...");

    var butteredToast = await ApplyButterAsync(toast);
    Console.WriteLine("Butter applied, applying jam...");

    var finalToast = await ApplyJamAsync(butteredToast);
    Console.WriteLine("Breakfast completed with async/await!");
}
```

Why async/await is preferred

The `async`/`await` approach offers several advantages:

- **Readability**: The code reads like synchronous code, making it easier to understand the flow of operations.
- **Maintainability**: Adding or removing steps in the sequence requires minimal code changes.
- **Error handling**: Exception handling with `try`/`catch` blocks works naturally, whereas `ContinueWith` requires careful handling of faulted tasks.
- **Debugging**: The call stack and debugger experience is much better with `async`/`await`.
- **Performance**: The compiler optimizations for `async`/`await` are more sophisticated than manual `ContinueWith` chains.

The benefit becomes even more apparent as the number of chained operations increases. While a single continuation might be manageable with `ContinueWith`, sequences of 3-4 or more asynchronous operations quickly become difficult to read and maintain. This pattern, known as "monadic do-notation" in functional programming, allows you to compose multiple asynchronous operations in a sequential, readable manner.

Next step

Explore real world scenarios for asynchronous programs

Asynchronous programming scenarios

08/15/2025

If your code implements I/O-bound scenarios to support network data requests, database access, or file system read/writes, asynchronous programming is the best approach. You can also write asynchronous code for CPU-bound scenarios like expensive calculations.

C# has a language-level asynchronous programming model that allows you to easily write asynchronous code without having to juggle callbacks or conform to a library that supports asynchrony. The model follows what is known as the Task-based asynchronous pattern (TAP).

Explore the asynchronous programming model

The `Task` and `Task<T>` objects represent the core of asynchronous programming. These objects are used to model asynchronous operations by supporting the `async` and `await` keywords. In most cases, the model is fairly simple for both I/O-bound and CPU-bound scenarios. Inside an `async` method:

- **I/O-bound code** starts an operation represented by a `Task` or `Task<T>` object within the `async` method.
- **CPU-bound code** starts an operation on a background thread with the Task.Run method.

In both cases, an active `Task` represents an asynchronous operation that might not be complete.

The `await` keyword is where the magic happens. It yields control to the caller of the method that contains the `await` expression, and ultimately allows the UI to be responsive or a service to be elastic. While there are ways to approach asynchronous code other than by using the `async` and `await` expressions, this article focuses on the language-level constructs.

> **Note**
>
> Some examples presented in this article use the System.Net.Http.HttpClient class to download data from a web service. In the example code, the `s_httpClient` object is a static field of type `Program` class:
>
> ```
> private static readonly HttpClient s_httpClient = new();
> ```
>
> For more information, see the complete example code at the end of this article.

Review underlying concepts

When you implement asynchronous programming in your C# code, the compiler transforms your program into a state machine. This construct tracks various operations and state in your code, such as yielding execution when the code reaches an `await` expression, and resuming execution when a background job completes.

In terms of computer science theory, asynchronous programming is an implementation of the Promise model of asynchrony.

In the asynchronous programming model, there are several key concepts to understand:

- You can use asynchronous code for both I/O-bound and CPU-bound code, but the implementation is different.
- Asynchronous code uses `Task<T>` and `Task` objects as constructs to model work running in the background.
- The `async` keyword declares a method as an asynchronous method, which allows you to use the `await` keyword in the method body.
- When you apply the `await` keyword, the code suspends the calling method and yields control back to its caller until the task completes.
- You can only use the `await` expression in an asynchronous method.

I/O-bound example: Download data from web service

In this example, when the user selects a button, the app downloads data from a web service. You don't want to block the UI thread for the app during the download process. The following code accomplishes this task:

C#

```
s_downloadButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI as the request
    // from the web service is happening.
    //
    // The UI thread is now free to perform other work.
    var stringData = await s_httpClient.GetStringAsync(URL);
    DoSomethingWithData(stringData);
};
```

The code expresses the intent (downloading data asynchronously) without getting bogged down in interacting with `Task` objects.

CPU-bound example: Run game calculation

In the next example, a mobile game inflicts damage on several agents on the screen in response to a button event. Performing the damage calculation can be expensive. Running the calculation on the UI thread can cause display and UI interaction issues during the calculation.

The best way to handle the task is to start a background thread to complete the work with the `Task.Run` method. The operation yields by using an `await` expression. The operation resumes when the task completes. This approach allows the UI to run smoothly while the work completes in the background.

C#

```
static DamageResult CalculateDamageDone()
{
    return new DamageResult()
    {
        // Code omitted:
        //
        // Does an expensive calculation and returns
        // the result of that calculation.
    };
}

s_calculateButton.Clicked += async (o, e) =>
{
    // This line will yield control to the UI while CalculateDamageDone()
    // performs its work. The UI thread is free to perform other work.
    var damageResult = await Task.Run(() => CalculateDamageDone());
    DisplayDamage(damageResult);
};
```

The code clearly expresses the intent of the button `Clicked` event. It doesn't require managing a background thread manually, and it completes the task in a nonblocking manner.

Recognize CPU-bound and I/O-bound scenarios

The previous examples demonstrate how to use the `async` modifier and `await` expression for I/O-bound and CPU-bound work. An example for each scenario showcases how the code is different based on where the operation is bound. To prepare for your implementation, you need to understand how to identify when an operation is I/O-bound or CPU-bound. Your implementation choice can greatly affect the performance of your code and potentially lead to misusing constructs.

There are two primary questions to address before you write any code:

Expand table

Question	Scenario	Implementation
Should the code wait for a result or action, such as data from a database?	**I/O-bound**	Use the `async` modifier and `await` expression *without* the `Task.Run` method. Avoid using the Task Parallel Library.
Should the code run an expensive computation?	**CPU-bound**	Use the `async` modifier and `await` expression, but spawn off the work on another thread with the `Task.Run` method. This approach addresses concerns with CPU responsiveness. If the work is appropriate for concurrency and parallelism, also consider using the Task Parallel Library.

Always measure the execution of your code. You might discover that your CPU-bound work isn't costly enough compared with the overhead of context switches when multithreading. Every choice has tradeoffs. Pick the correct tradeoff for your situation.

Explore other examples

The examples in this section demonstrate several ways you can write asynchronous code in C#. They cover a few scenarios you might encounter.

Extract data from a network

The following code downloads HTML from a given URL and counts the number of times the string ".NET" occurs in the HTML. The code uses ASP.NET to define a Web API controller method, which performs the task and returns the count.

> **Note**
>
> If you plan on doing HTML parsing in production code, don't use regular expressions. Use a parsing library instead.

C#

```
[HttpGet, Route("DotNetCount")]
static public async Task<int> GetDotNetCountAsync(string URL)
{
    // Suspends GetDotNetCountAsync() to allow the caller (the web server)
    // to accept another request, rather than blocking on this one.
    var html = await s_httpClient.GetStringAsync(URL);
```

```
    return Regex.Matches(html, @"\.NET").Count;
}
```

You can write similar code for a Universal Windows App and perform the counting task after a button press:

C#

```
private readonly HttpClient _httpClient = new HttpClient();

private async void OnSeeTheDotNetsButtonClick(object sender, RoutedEventArgs e)
{
    // Capture the task handle here so we can await the background task later.
    var getDotNetFoundationHtmlTask =
_httpClient.GetStringAsync("https://dotnetfoundation.org");

    // Any other work on the UI thread can be done here, such as enabling a
Progress Bar.
    // It's important to do the extra work here before the "await" call,
    // so the user sees the progress bar before execution of this method is
yielded.
    NetworkProgressBar.IsEnabled = true;
    NetworkProgressBar.Visibility = Visibility.Visible;

    // The await operator suspends OnSeeTheDotNetsButtonClick(), returning control
to its caller.
    // This action is what allows the app to be responsive and not block the UI
thread.
    var html = await getDotNetFoundationHtmlTask;
    int count = Regex.Matches(html, @"\.NET").Count;

    DotNetCountLabel.Text = $"Number of .NETs on dotnetfoundation.org: {count}";

    NetworkProgressBar.IsEnabled = false;
    NetworkProgressBar.Visibility = Visibility.Collapsed;
}
```

Wait for multiple tasks to complete

In some scenarios, the code needs to retrieve multiple pieces of data concurrently. The `Task` APIs provide methods that enable you to write asynchronous code that performs a nonblocking wait on multiple background jobs:

- Task.WhenAll method
- Task.WhenAny method

The following example shows how you might grab `User` object data for a set of `userId` objects.

C#

```
private static async Task<User> GetUserAsync(int userId)
{
    // Code omitted:
    //
    // Given a user Id {userId}, retrieves a User object corresponding
    // to the entry in the database with {userId} as its Id.

    return await Task.FromResult(new User() { id = userId });
}

private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = new List<Task<User>>();
    foreach (int userId in userIds)
    {
        getUserTasks.Add(GetUserAsync(userId));
    }

    return await Task.WhenAll(getUserTasks);
}
```

You can write this code more succinctly by using LINQ:

C#

```
private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int> userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
    return await Task.WhenAll(getUserTasks);
}
```

Although you write less code by using LINQ, exercise caution when mixing LINQ with asynchronous code. LINQ uses deferred (or lazy) execution, which means that without immediate evaluation, async calls don't happen until the sequence is enumerated.

The previous example is correct and safe, because it uses the Enumerable.ToArray method to immediately evaluate the LINQ query and store the tasks in an array. This approach ensures the `id => GetUserAsync(id)` calls execute immediately and all tasks start concurrently, just like the `foreach` loop approach. Always use Enumerable.ToArray or Enumerable.ToList when creating tasks with LINQ to ensure immediate execution and concurrent task execution. Here's an example that demonstrates using `ToList()` with `Task.WhenAny` to process tasks as they complete:

C#

```
private static async Task ProcessTasksAsTheyCompleteAsync(IEnumerable<int>
userIds)
{
    var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToList();

    while (getUserTasks.Count > 0)
    {
        Task<User> completedTask = await Task.WhenAny(getUserTasks);
        getUserTasks.Remove(completedTask);

        User user = await completedTask;
        Console.WriteLine($"Processed user {user.id}");
    }
}
```

In this example, `ToList()` creates a list that supports the `Remove()` operation, allowing you to dynamically remove completed tasks. This pattern is particularly useful when you want to handle results as soon as they're available, rather than waiting for all tasks to complete.

Although you write less code by using LINQ, exercise caution when mixing LINQ with asynchronous code. LINQ uses deferred (or lazy) execution. Asynchronous calls don't happen immediately as they do in a `foreach` loop, unless you force the generated sequence to iterate with a call to the `.ToList()` or `.ToArray()` method.

You can choose between Enumerable.ToArray and Enumerable.ToList based on your scenario:

- Use `ToArray()` when you plan to process all tasks together, such as with `Task.WhenAll`. Arrays are efficient for scenarios where the collection size is fixed.
- Use `ToList()` when you need to dynamically manage tasks, such as with `Task.WhenAny` where you might remove completed tasks from the collection as they finish.

Review considerations for asynchronous programming

With asynchronous programming, there are several details to keep in mind that can prevent unexpected behavior.

Use await inside async() method body

When you use the `async` modifier, you should include one or more `await` expressions in the method body. If the compiler doesn't encounter an `await` expression, the method fails to yield. Although the compiler generates a warning, the code still compiles and the compiler runs the

method. The state machine generated by the C# compiler for the asynchronous method doesn't accomplish anything, so the entire process is highly inefficient.

Add "Async" suffix to asynchronous method names

The .NET style convention is to add the "Async" suffix to all asynchronous method names. This approach helps to more easily differentiate between synchronous and asynchronous methods. Certain methods that aren't explicitly called by your code (such as event handlers or web controller methods) don't necessarily apply in this scenario. Because these items aren't explicitly called by your code, using explicit naming isn't as important.

Return 'async void' only from event handlers

Event handlers must declare `void` return types and can't use or return `Task` and `Task<T>` objects as other methods do. When you write asynchronous event handlers, you need to use the `async` modifier on a `void` returning method for the handlers. Other implementations of `async void` returning methods don't follow the TAP model and can present challenges:

- Exceptions thrown in an `async void` method can't be caught outside of that method
- `async void` methods are difficult to test
- `async void` methods can cause negative side effects if the caller isn't expecting them to be asynchronous

Use caution with asynchronous lambdas in LINQ

It's important to use caution when you implement asynchronous lambdas in LINQ expressions. Lambda expressions in LINQ use deferred execution, which means the code can execute at an unexpected time. The introduction of blocking tasks into this scenario can easily result in a deadlock, if the code isn't written correctly. Moreover, the nesting of asynchronous code can also make it difficult to reason about the execution of the code. Async and LINQ are powerful, but these techniques should be used together as carefully and clearly as possible.

Yield for tasks in a nonblocking manner

If your program needs the result of a task, write code that implements the `await` expression in a nonblocking manner. Blocking the current thread as a means to wait synchronously for a `Task` item to complete can result in deadlocks and blocked context threads. This programming approach can require more complex error-handling. The following table provides guidance on how access results from tasks in a nonblocking way:

Expand table

Task scenario	Current code	Replace with 'await'
Retrieve the result of a background task	`Task.Wait` or `Task.Result`	`await`
Continue when any task completes	`Task.WaitAny`	`await Task.WhenAny`
*Continue when **all** tasks complete*	`Task.WaitAll`	`await Task.WhenAll`
Continue after some amount of time	`Thread.Sleep`	`await Task.Delay`

Consider using ValueTask type

When an asynchronous method returns a `Task` object, performance bottlenecks might be introduced in certain paths. Because `Task` is a reference type, a `Task` object is allocated from the heap. If a method declared with the `async` modifier returns a cached result or completes synchronously, the extra allocations can accrue significant time costs in performance critical sections of code. This scenario can become costly when the allocations occur in tight loops. For more information, see generalized async return types.

Understand when to set ConfigureAwait(false)

Developers often inquire about when to use the Task.ConfigureAwait(Boolean) boolean. This API allows for a `Task` instance to configure the context for the state machine that implements any `await` expression. When the boolean isn't set correctly, performance can degrade or deadlocks can occur. For more information, see ConfigureAwait FAQ.

Write less-stateful code

Avoid writing code that depends on the state of global objects or the execution of certain methods. Instead, depend only on the return values of methods. There are many benefits to writing code that is less-stateful:

- Easier to reason about code
- Easier to test code
- More simple to mix asynchronous and synchronous code
- Able to avoid race conditions in code
- Simple to coordinate asynchronous code that depends on return values
- (Bonus) Works well with dependency injection in code

A recommended goal is to achieve complete or near-complete Referential Transparency in your code. This approach results in a predictable, testable, and maintainable codebase.

Synchronous access to asynchronous operations

In scenarios, you might need to block on asynchronous operations when the `await` keyword isn't available throughout your call stack. This situation occurs in legacy codebases or when integrating asynchronous methods into synchronous APIs that can't be changed.

> **⚠ Warning**
>
> Synchronous blocking on asynchronous operations can lead to deadlocks and should be avoided whenever possible. The preferred solution is to use `async`/`await` throughout your call stack.

When you must block synchronously on a `Task`, here are the available approaches, listed from most to least preferred:

- Use GetAwaiter().GetResult()
- Use Task.Run for complex scenarios
- Use Wait() and Result

Use GetAwaiter().GetResult()

The `GetAwaiter().GetResult()` pattern is generally the preferred approach when you must block synchronously:

```
C#
```

```
// When you cannot use await
Task<string> task = GetDataAsync();
string result = task.GetAwaiter().GetResult();
```

This approach:

- Preserves the original exception without wrapping it in an `AggregateException`.
- Blocks the current thread until the task completes.
- Still carries deadlock risk if not used carefully.

Use Task.Run for complex scenarios

For complex scenarios where you need to isolate the asynchronous work:

```
C#

// Offload to thread pool to avoid context deadlocks
string result = Task.Run(async () => await
GetDataAsync()).GetAwaiter().GetResult();
```

This pattern:

- Executes the asynchronous method on a thread pool thread.
- Can help avoid some deadlock scenarios.
- Adds overhead by scheduling work to the thread pool.

Use Wait() and Result

You can use a blocking approach by calling Wait() and Result. However, this approach is discouraged because it wraps exceptions in AggregateException.

```
C#

Task<string> task = GetDataAsync();
task.Wait();
string result = task.Result;
```

Problems with `Wait()` and `Result`:

- Exceptions are wrapped in `AggregateException`, making error handling more complex.
- Higher deadlock risk.
- Less clear intent in code.

Additional considerations

- Deadlock prevention: Be especially careful in UI applications or when using a synchronization context.
- Performance impact: Blocking threads reduces scalability.
- Exception handling: Test error scenarios carefully as exception behavior differs between patterns.

For more detailed guidance on the challenges and considerations of synchronous wrappers for asynchronous methods, see Should I expose synchronous wrappers for asynchronous methods? .

Review the complete example

The following code represents the complete example, which is available in the *Program.cs* example file.

C#

```
using System.Text.RegularExpressions;
using System.Windows;
using Microsoft.AspNetCore.Mvc;

class Button
{
    public Func<object, object, Task>? Clicked
    {
        get;
        internal set;
    }
}

class DamageResult
{
    public int Damage
    {
        get { return 0; }
    }
}

class User
{
    public bool isEnabled
    {
        get;
        set;
    }

    public int id
    {
        get;
        set;
    }
}

public class Program
{
    private static readonly Button s_downloadButton = new();
    private static readonly Button s_calculateButton = new();

    private static readonly HttpClient s_httpClient = new();

    private static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
```

```
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-
app-visual-studio",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/shows/net-core-101/what-is-net",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://dotnetfoundation.org",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/maui"
    };

    private static void Calculate()
    {
        static DamageResult CalculateDamageDone()
        {
            return new DamageResult()
            {
                // Code omitted:
                //
                // Does an expensive calculation and returns
                // the result of that calculation.
            };
        }

        s_calculateButton.Clicked += async (o, e) =>
        {
            // This line will yield control to the UI while CalculateDamageDone()
            // performs its work. The UI thread is free to perform other work.
            var damageResult = await Task.Run(() => CalculateDamageDone());
            DisplayDamage(damageResult);
        };
    }

    private static void DisplayDamage(DamageResult damage)
    {
        Console.WriteLine(damage.Damage);
    }

    private static void Download(string URL)
    {
        s_downloadButton.Clicked += async (o, e) =>
        {
            // This line will yield control to the UI as the request
            // from the web service is happening.
```

```
            //
            // The UI thread is now free to perform other work.
            var stringData = await s_httpClient.GetStringAsync(URL);
            DoSomethingWithData(stringData);
        };
    }

    private static void DoSomethingWithData(object stringData)
    {
        Console.WriteLine($"Displaying data: {stringData}");
    }

    private static async Task<User> GetUserAsync(int userId)
    {
        // Code omitted:
        //
        // Given a user Id {userId}, retrieves a User object corresponding
        // to the entry in the database with {userId} as its Id.

        return await Task.FromResult(new User() { id = userId });
    }

    private static async Task<IEnumerable<User>> GetUsersAsync(IEnumerable<int>
userIds)
    {
        var getUserTasks = new List<Task<User>>();
        foreach (int userId in userIds)
        {
            getUserTasks.Add(GetUserAsync(userId));
        }

        return await Task.WhenAll(getUserTasks);
    }

    private static async Task<User[]> GetUsersAsyncByLINQ(IEnumerable<int>
userIds)
    {
        var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToArray();
        return await Task.WhenAll(getUserTasks);
    }

    private static async Task ProcessTasksAsTheyCompleteAsync(IEnumerable<int>
userIds)
    {
        var getUserTasks = userIds.Select(id => GetUserAsync(id)).ToList();

        while (getUserTasks.Count > 0)
        {
            Task<User> completedTask = await Task.WhenAny(getUserTasks);
            getUserTasks.Remove(completedTask);

            User user = await completedTask;
            Console.WriteLine($"Processed user {user.id}");
        }
    }
```

```
    [HttpGet, Route("DotNetCount")]
    static public async Task<int> GetDotNetCountAsync(string URL)
    {
        // Suspends GetDotNetCountAsync() to allow the caller (the web server)
        // to accept another request, rather than blocking on this one.
        var html = await s_httpClient.GetStringAsync(URL);
        return Regex.Matches(html, @"\.NET").Count;
    }

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        Console.WriteLine("Counting '.NET' phrase in websites...");
        int total = 0;
        foreach (string url in s_urlList)
        {
            var result = await GetDotNetCountAsync(url);
            Console.WriteLine($"{url}: {result}");
            total += result;
        }
        Console.WriteLine("Total: " + total);

        Console.WriteLine("Retrieving User objects with list of IDs...");
        IEnumerable<int> ids = new int[] { 1, 2, 3, 4, 5, 6, 7, 8, 9, 0 };
        var users = await GetUsersAsync(ids);
        foreach (User? user in users)
        {
            Console.WriteLine($"{user.id}: isEnabled={user.isEnabled}");
        }

        Console.WriteLine("Processing tasks as they complete...");
        await ProcessTasksAsTheyCompleteAsync(ids);

        Console.WriteLine("Application ending.");
    }
}

// Example output:
//
// Application started.
// Counting '.NET' phrase in websites...
// https://learn.microsoft.com: 0
// https://learn.microsoft.com/aspnet/core: 57
// https://learn.microsoft.com/azure: 1
// https://learn.microsoft.com/azure/devops: 2
// https://learn.microsoft.com/dotnet: 83
// https://learn.microsoft.com/dotnet/desktop/wpf/get-started/create-app-visual-
studio: 31
// https://learn.microsoft.com/education: 0
// https://learn.microsoft.com/shows/net-core-101/what-is-net: 42
// https://learn.microsoft.com/enterprise-mobility-security: 0
// https://learn.microsoft.com/gaming: 0
// https://learn.microsoft.com/graph: 0
```

```
// https://learn.microsoft.com/microsoft-365: 0
// https://learn.microsoft.com/office: 0
// https://learn.microsoft.com/powershell: 0
// https://learn.microsoft.com/sql: 0
// https://learn.microsoft.com/surface: 0
// https://dotnetfoundation.org: 16
// https://learn.microsoft.com/visualstudio: 0
// https://learn.microsoft.com/windows: 0
// https://learn.microsoft.com/maui: 6
// Total: 238
// Retrieving User objects with list of IDs...
// 1: isEnabled= False
// 2: isEnabled= False
// 3: isEnabled= False
// 4: isEnabled= False
// 5: isEnabled= False
// 6: isEnabled= False
// 7: isEnabled= False
// 8: isEnabled= False
// 9: isEnabled= False
// 0: isEnabled= False
// Application ending.
```

Related links

- The Task asynchronous programming model (C#)

Task asynchronous programming model

10/13/2025

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# supports simplified approach, async programming, that uses asynchronous support in the .NET 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 article provides an overview of when and how to use async programming and includes links to other articles 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.

Expand table

Application area	.NET types with async methods	Windows Runtime types with async methods
Web access	HttpClient	Windows.Web.Http.HttpClient SyndicationClient
Working with files	JsonSerializer StreamReader StreamWriter XmlReader XmlWriter	StorageFile

Application area	.NET types with async methods	Windows Runtime types with async methods
Working with images		MediaCapture BitmapEncoder BitmapDecoder
WCF programming	Synchronous and Asynchronous Operations	

Asynchrony proves especially valuable for applications that access the UI thread because all UI-related activity usually shares one thread. If any process is blocked in a synchronous application, all are blocked. Your application stops responding, and you might conclude that it failed when instead it's just waiting.

When you use asynchronous methods, the application continues to respond to the UI. You can resize or minimize a window, for example, or you can close the application if you don't want to wait for it to finish.

The async-based approach adds the equivalent of an automatic transmission to the list of options that you can choose from when designing asynchronous operations. That is, you get all the benefits of traditional asynchronous programming but with much less effort from the developer.

Async methods are easy to write

The async and await keywords in C# are the heart of async programming. By using those two keywords, you can use resources in .NET Framework, .NET Core, or the Windows Runtime to create an asynchronous method almost as easily as you create a synchronous method. Asynchronous methods that you define by using the `async` keyword are referred to as *async methods*.

The following example shows an async method. Almost everything in the code should look familiar to you.

You can find a complete Windows Presentation Foundation (WPF) example available for download from Asynchronous programming with async and await in C#.

C#

```
public async Task<int> GetUrlContentLengthAsync()
{
    using var client = new HttpClient();

    Task<string> getStringTask =
```

```
        client.GetStringAsync("https://learn.microsoft.com/dotnet");

    DoIndependentWork();

    string contents = await getStringTask;

    return contents.Length;
}

void DoIndependentWork()
{
    Console.WriteLine("Working...");
}
```

You can learn several practices from the preceding sample. Start with the method signature. It includes the `async` modifier. The return type is `Task<int>` (See "Return Types" section for more options). The method name ends in `Async`. In the body of the method, `GetStringAsync` returns a `Task<string>`. That means that when you `await` the task you get a `string` (`contents`). Before awaiting the task, you can do work that doesn't rely on the `string` from `GetStringAsync`.

Pay close attention to the `await` operator. It suspends `GetUrlContentLengthAsync`:

- `GetUrlContentLengthAsync` can't continue until `getStringTask` is complete.
- Meanwhile, control returns to the caller of `GetUrlContentLengthAsync`.
- Control resumes here when `getStringTask` is complete.
- The `await` operator then retrieves the `string` result from `getStringTask`.

The return statement specifies an integer result. Any methods that are awaiting `GetUrlContentLengthAsync` retrieve the length value.

If `GetUrlContentLengthAsync` doesn't have any work that it can do between calling `GetStringAsync` and awaiting its completion, you can simplify your code by calling and awaiting in the following single statement.

C#

```
string contents = await
client.GetStringAsync("https://learn.microsoft.com/dotnet");
```

The following characteristics summarize what makes the previous example an async method:

- The method signature includes an `async` modifier.
- The name of an async method, by convention, ends with an "Async" suffix.
- The return type is one of the following types:

- Task<TResult> if your method has a return statement in which the operand has type `TResult`.
- Task if your method has no return statement or has a return statement with no operand.
- `void` if you're writing an async event handler.
- Any other type that has a `GetAwaiter` method.

For more information, see the Return types and parameters section.

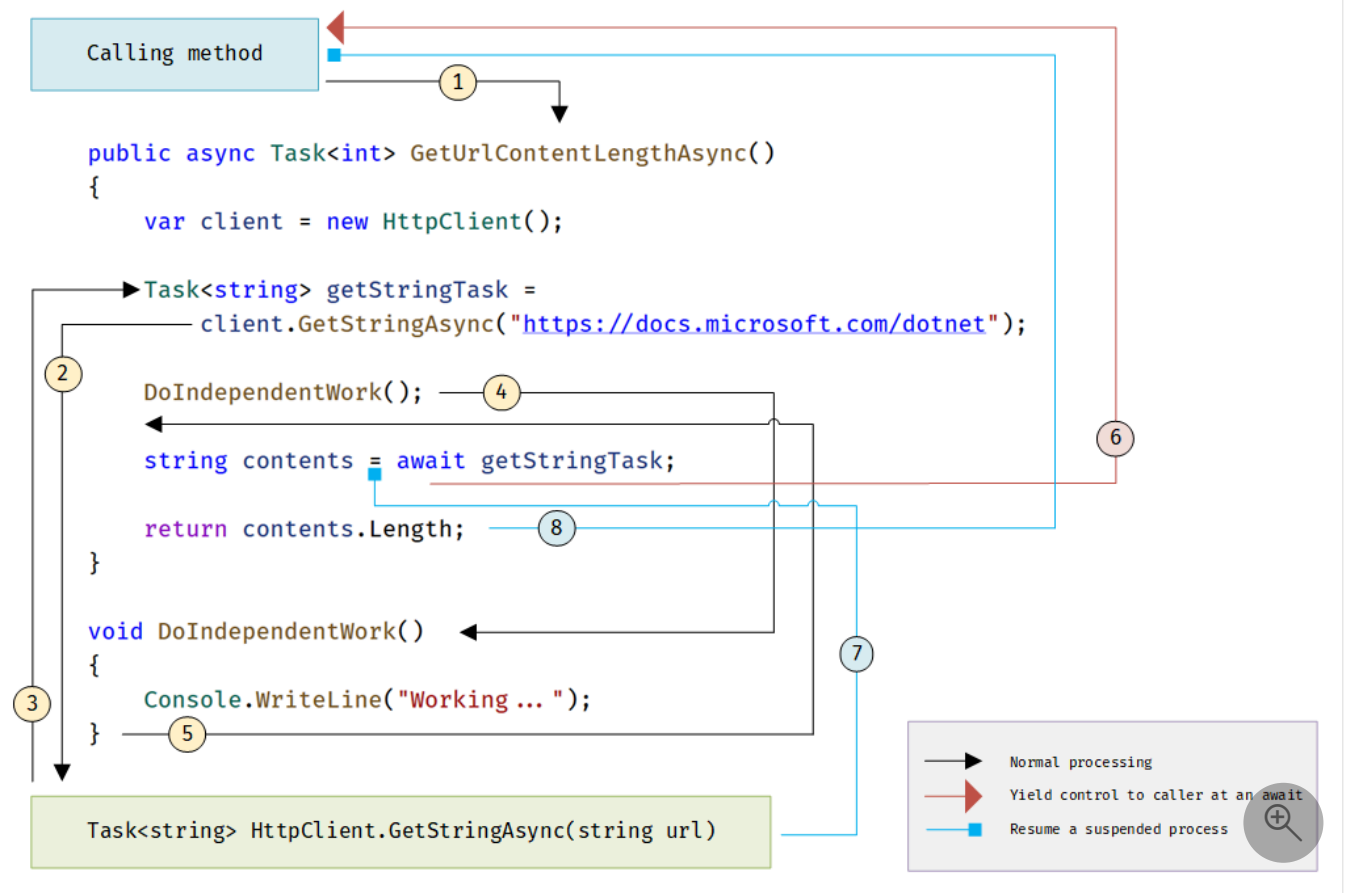
- The method usually includes at least one `await` expression, which marks a point where the method can't continue until the awaited asynchronous operation is complete. In the meantime, the method is suspended, and control returns to the method's caller. The next section of this article illustrates what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

1. A calling method calls and awaits the `GetUrlContentLengthAsync` async method.

2. `GetUrlContentLengthAsync` creates an HttpClient instance and calls the GetStringAsync asynchronous method to download the contents of a website as a string.

3. Something happens in `GetStringAsync` that suspends its progress. Perhaps it must wait for a website to download or some other blocking activity. To avoid blocking resources, `GetStringAsync` yields control to its caller, `GetUrlContentLengthAsync`.

   `GetStringAsync` returns a Task<TResult>, where `TResult` is a string, and `GetUrlContentLengthAsync` assigns the task to the `getStringTask` variable. The task represents the ongoing process for the call to `GetStringAsync`, with a commitment to produce an actual string value when the work is complete.

4. Because `getStringTask` isn't awaited yet, `GetUrlContentLengthAsync` can continue with other work that doesn't depend on the final result from `GetStringAsync`. That work is represented by a call to the synchronous method `DoIndependentWork`.

5. `DoIndependentWork` is a synchronous method that does its work and returns to its caller.

6. `GetUrlContentLengthAsync` runs out of work that it can do without a result from `getStringTask`. `GetUrlContentLengthAsync` next wants to calculate and return the length of the downloaded string, but the method can't calculate that value until the method has the string.

   Therefore, `GetUrlContentLengthAsync` uses an await operator to suspend its progress and to yield control to the method that called `GetUrlContentLengthAsync`. `GetUrlContentLengthAsync` returns a `Task<int>` to the caller. The task represents a promise to produce an integer result that's the length of the downloaded string.

   > **Note**
   >
   > If `GetStringAsync` (and therefore `getStringTask`) completes before `GetUrlContentLengthAsync` awaits it, control remains in `GetUrlContentLengthAsync`. The expense of suspending and then returning to `GetUrlContentLengthAsync` would be wasted if the called asynchronous process `getStringTask` is complete and `GetUrlContentLengthAsync` doesn't have to wait for the final result.

   Inside the calling method the processing pattern continues. The caller might do other work that doesn't depend on the result from `GetUrlContentLengthAsync` before awaiting that result, or the caller might await immediately. The calling method is waiting for `GetUrlContentLengthAsync`, and `GetUrlContentLengthAsync` is waiting for `GetStringAsync`.

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 `contents`.

8. When `GetUrlContentLengthAsync` has the string result, the method can calculate the length of the string. Then the work of `GetUrlContentLengthAsync` is also complete, and the waiting event handler can resume. In the full example at the end of the article, you can confirm that the event handler retrieves and prints the value of the length result. If you're 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.

API async methods

You might be wondering where to find methods such as `GetStringAsync` that support async programming. .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 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 extra 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 thread pool.

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 articles:

- 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.

You can also specify any other return type, if the type includes a `GetAwaiter` method. ValueTask<TResult> is an example of such a type. It's available in the System.Threading.Tasks.Extension NuGet package.

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

C#

```
async Task<int> GetTaskOfTResultAsync()
{
    int hours = 0;
```

```
    await Task.Delay(0);

    return hours;
}


Task<int> returnedTaskTResult = GetTaskOfTResultAsync();
int intResult = await returnedTaskTResult;
// Single line
// int intResult = await GetTaskOfTResultAsync();

async Task GetTaskAsync()
{
    await Task.Delay(0);
    // No return statement needed
}

Task returnedTask = GetTaskAsync();
await returnedTask;
// Single line
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#).

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, methods that return commonly awaitable types (for example, `Task`, `Task<T>`, `ValueTask`, `ValueTask<T>`) should have names that end with "Async". Methods that start an asynchronous operation but don't return an awaitable type shouldn't have names that end with "Async", but might start with "Begin", "Start", or some other verb to suggest this method doesn't return or throw the result of the operation.

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 `OnButtonClick`.

Related articles (Visual Studio)

Expand table

Title	Description
How to make multiple web requests in parallel by using async and await (C#)	Demonstrates how to start several tasks at the same time.
Async return types (C#)	Illustrates the types that async methods can return, and explains when each type is appropriate.
Cancel tasks with a cancellation token as a signaling mechanism.	Shows how to add the following functionality to your async solution: - Cancel a list of tasks (C#) - Cancel tasks after a period of time (C#) - Process asynchronous task as they complete (C#)
Using async for file access (C#)	Lists and demonstrates the benefits of using async and await to access files.
Task-based asynchronous pattern (TAP)	Describes an asynchronous pattern. The pattern is based on the Task and Task<TResult> types.
Async Videos on Channel 9	Provides links to various videos about async programming.

See also

- Asynchronous programming with async and await
- async

- await

Async return types (C#)

Article • 11/22/2024

Async methods can have the following return types:

- Task, for an async method that performs an operation but returns no value.
- Task<TResult>, for an async method that returns a value.
- `void`, for an event handler.
- Any type that has an accessible `GetAwaiter` method. The object returned by the `GetAwaiter` method must implement the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface.
- IAsyncEnumerable<T>, for an async method that returns an *async stream*.

For more information about async methods, see Asynchronous programming with async and await (C#).

Several other types also exist that are specific to Windows workloads:

- DispatcherOperation, for async operations limited to Windows.
- IAsyncAction, for async actions in Universal Windows Platform (UWP) apps that don't return a value.
- IAsyncActionWithProgress<TProgress>, for async actions in UWP apps that report progress but don't return a value.
- IAsyncOperation<TResult>, for async operations in UWP apps that return a value.
- IAsyncOperationWithProgress<TResult,TProgress>, for async operations in UWP apps that report progress and return a value.

Task return type

Async methods that don't contain a `return` statement or that contain a `return` statement that doesn't return an operand usually have a return type of Task. Such methods return `void` if they run synchronously. If you use a Task return type for an async method, a calling method can use an `await` operator to suspend the caller's completion until the called async method finishes.

In the following example, the `WaitAndApologizeAsync` method doesn't contain a `return` statement, so the method returns a Task object. Returning a `Task` enables `WaitAndApologizeAsync` to be awaited. The Task type doesn't include a `Result` property because it has no return value.

```
C#
```

```
public static async Task DisplayCurrentInfoAsync()
{
    await WaitAndApologizeAsync();

    Console.WriteLine($"Today is {DateTime.Now:D}");
    Console.WriteLine($"The current time is {DateTime.Now.TimeOfDay:t}");
    Console.WriteLine("The current temperature is 76 degrees.");
}

static async Task WaitAndApologizeAsync()
{
    await Task.Delay(2000);

    Console.WriteLine("Sorry for the delay...\n");
}
// Example output:
//    Sorry for the delay...
//
// Today is Monday, August 17, 2020
// The current time is 12:59:24.2183304
// The current temperature is 76 degrees.
```

`WaitAndApologizeAsync` is awaited by using an await statement instead of an await expression, similar to the calling statement for a synchronous void-returning method. The application of an await operator in this case doesn't produce a value. When the right operand of an `await` is a Task<TResult>, the `await` expression produces a result of `T`. When the right operand of an `await` is a Task, the `await` and its operand are a statement.

You can separate the call to `WaitAndApologizeAsync` from the application of an await operator, as the following code shows. However, remember that a `Task` doesn't have a `Result` property, and that no value is produced when an await operator is applied to a `Task`.

The following code separates calling the `WaitAndApologizeAsync` method from awaiting the task that the method returns.

C#

```
Task waitAndApologizeTask = WaitAndApologizeAsync();

string output =
    $"Today is {DateTime.Now:D}\n" +
    $"The current time is {DateTime.Now.TimeOfDay:t}\n" +
    "The current temperature is 76 degrees.\n";
```

```
await waitAndApologizeTask;
Console.WriteLine(output);
```

Task<TResult> return type

The Task<TResult> return type is used for an async method that contains a return statement in which the operand is `TResult`.

In the following example, the `GetLeisureHoursAsync` method contains a `return` statement that returns an integer. The method declaration must specify a return type of `Task<int>`. The FromResult async method is a placeholder for an operation that returns a DayOfWeek.

C#

```
public static async Task ShowTodaysInfoAsync()
{
    string message =
        $"Today is {DateTime.Today:D}\n" +
        "Today's hours of leisure: " +
        $"{await GetLeisureHoursAsync()}";

    Console.WriteLine(message);
}

static async Task<int> GetLeisureHoursAsync()
{
    DayOfWeek today = await Task.FromResult(DateTime.Now.DayOfWeek);

    int leisureHours =
        today is DayOfWeek.Saturday || today is DayOfWeek.Sunday
        ? 16 : 5;

    return leisureHours;
}
// Example output:
//    Today is Wednesday, May 24, 2017
//    Today's hours of leisure: 5
```

When `GetLeisureHoursAsync` is called from within an await expression in the `ShowTodaysInfo` method, the await expression retrieves the integer value (the value of `leisureHours`) stored in the task returned by the `GetLeisureHours` method. For more information about await expressions, see await.

You can better understand how `await` retrieves the result from a `Task<T>` by separating the call to `GetLeisureHoursAsync` from the application of `await`, as the following code

shows. A call to method `GetLeisureHoursAsync` that isn't immediately awaited returns a `Task<int>`, as you would expect from the declaration of the method. The task is assigned to the `getLeisureHoursTask` variable in the example. Because `getLeisureHoursTask` is a Task<TResult>, it contains a Result property of type `TResult`. In this case, `TResult` represents an integer type. When `await` is applied to `getLeisureHoursTask`, the await expression evaluates to the contents of the Result property of `getLeisureHoursTask`. The value is assigned to the `ret` variable.

> **ⓘ Important**
>
> The **Result** property is a blocking property. If you try to access it before its task is finished, the thread that's currently active is blocked until the task completes and the value is available. In most cases, you should access the value by using `await` instead of accessing the property directly.
>
> The previous example retrieved the value of the **Result** property to block the main thread so that the `Main` method could print the `message` to the console before the application ended.

C#

```
var getLeisureHoursTask = GetLeisureHoursAsync();

string message =
    $"Today is {DateTime.Today:D}\n" +
    "Today's hours of leisure: " +
    $"{await getLeisureHoursTask}";

Console.WriteLine(message);
```

Void return type

You use the `void` return type in asynchronous event handlers, which require a `void` return type. For methods other than event handlers that don't return a value, you should return a Task instead, because an async method that returns `void` can't be awaited. Any caller of such a method must continue to completion without waiting for the called async method to finish. The caller must be independent of any values or exceptions that the async method generates.

The caller of a void-returning async method can't catch exceptions thrown from the method. Such unhandled exceptions are likely to cause your application to fail. If a

method that returns a Task or Task<TResult> throws an exception, the exception is stored in the returned task. The exception is rethrown when the task is awaited. Make sure that any async method that can produce an exception has a return type of Task or Task<TResult> and that calls to the method are awaited.

The following example shows the behavior of an async event handler. In the example code, an async event handler must let the main thread know when it finishes. Then the main thread can wait for an async event handler to complete before exiting the program.

C#

```
public class NaiveButton
{
    public event EventHandler? Clicked;

    public void Click()
    {
        Console.WriteLine("Somebody has clicked a button. Let's raise the
event...");
        Clicked?.Invoke(this, EventArgs.Empty);
        Console.WriteLine("All listeners are notified.");
    }
}

public class AsyncVoidExample
{
    static readonly TaskCompletionSource<bool> s_tcs = new
TaskCompletionSource<bool>();

    public static async Task MultipleEventHandlersAsync()
    {
        Task<bool> secondHandlerFinished = s_tcs.Task;

        var button = new NaiveButton();

        button.Clicked += OnButtonClicked1;
        button.Clicked += OnButtonClicked2Async;
        button.Clicked += OnButtonClicked3;

        Console.WriteLine("Before button.Click() is called...");
        button.Click();
        Console.WriteLine("After button.Click() is called...");

        await secondHandlerFinished;
    }

    private static void OnButtonClicked1(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 1 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 1 is done.");
```

```
    }

    private static async void OnButtonClicked2Async(object? sender,
EventArgs e)
    {
        Console.WriteLine("   Handler 2 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 2 is about to go async...");
        await Task.Delay(500);
        Console.WriteLine("   Handler 2 is done.");
        s_tcs.SetResult(true);
    }

    private static void OnButtonClicked3(object? sender, EventArgs e)
    {
        Console.WriteLine("   Handler 3 is starting...");
        Task.Delay(100).Wait();
        Console.WriteLine("   Handler 3 is done.");
    }
}
// Example output:
//
// Before button.Click() is called...
// Somebody has clicked a button. Let's raise the event...
//    Handler 1 is starting...
//    Handler 1 is done.
//    Handler 2 is starting...
//    Handler 2 is about to go async...
//    Handler 3 is starting...
//    Handler 3 is done.
// All listeners are notified.
// After button.Click() is called...
//    Handler 2 is done.
```

Generalized async return types and ValueTask<TResult>

An async method can return any type that has an accessible `GetAwaiter` method that returns an instance of an *awaiter type*. In addition, the type returned from the `GetAwaiter` method must have the System.Runtime.CompilerServices.AsyncMethodBuilderAttribute attribute. You can learn more in the article on Attributes read by the compiler or the C# spec for the Task type builder pattern.

This feature is the complement to awaitable expressions, which describes the requirements for the operand of `await`. Generalized async return types enable the compiler to generate `async` methods that return different types. Generalized async return types enabled performance improvements in the .NET libraries. Because Task and

Task<TResult> are reference types, memory allocation in performance-critical paths, particularly when allocations occur in tight loops, can adversely affect performance. Support for generalized return types means that you can return a lightweight value type instead of a reference type to avoid more memory allocations.

.NET provides the System.Threading.Tasks.ValueTask<TResult> structure as a lightweight implementation of a generalized task-returning value. The following example uses the ValueTask<TResult> structure to retrieve the value of two dice rolls.

C#

```
class Program
{
    static readonly Random s_rnd = new Random();

    static async Task Main() =>
        Console.WriteLine($"You rolled {await GetDiceRollAsync()}");

    static async ValueTask<int> GetDiceRollAsync()
    {
        Console.WriteLine("Shaking dice...");

        int roll1 = await RollAsync();
        int roll2 = await RollAsync();

        return roll1 + roll2;
    }

    static async ValueTask<int> RollAsync()
    {
        await Task.Delay(500);

        int diceRoll = s_rnd.Next(1, 7);
        return diceRoll;
    }
}
// Example output:
//    Shaking dice...
//    You rolled 8
```

Writing a generalized async return type is an advanced scenario, and is targeted for use in specialized environments. Consider using the `Task`, `Task<T>`, and `ValueTask<T>` types instead, which cover most scenarios for asynchronous code.

You can apply the `AsyncMethodBuilder` attribute to an async method (instead of the async return type declaration) to override the builder for that type. Typically you'd apply this attribute to use a different builder provided in the .NET runtime.

Async streams with IAsyncEnumerable<T>

An async method might return an *async stream*, represented by IAsyncEnumerable<T>. An async stream provides a way to enumerate items read from a stream when elements are generated in chunks with repeated asynchronous calls. The following example shows an async method that generates an async stream:

C#

```
static async IAsyncEnumerable<string> ReadWordsFromStreamAsync()
{
    string data =
        @"This is a line of text.
              Here is the second line of text.
              And there is one more for good measure.
              Wait, that was the penultimate line.";

    using var readStream = new StringReader(data);

    string? line = await readStream.ReadLineAsync();
    while (line != null)
    {
        foreach (string word in line.Split(' ',
StringSplitOptions.RemoveEmptyEntries))
        {
            yield return word;
        }

        line = await readStream.ReadLineAsync();
    }
}
```

The preceding example reads lines from a string asynchronously. Once each line is read, the code enumerates each word in the string. Callers would enumerate each word using the `await foreach` statement. The method awaits when it needs to asynchronously read the next line from the source string.

See also

- FromResult
- Process asynchronous tasks as they complete
- Asynchronous programming with async and await (C#)
- async
- await

Process asynchronous tasks as they complete (C#)

09/09/2025

By using Task.WhenAny, you can start multiple tasks at the same time and process them one by one as they're completed rather than process them in the order in which they're started.

The following example uses a query to create a collection of tasks. Each task downloads the contents of a specified website. In each iteration of a while loop, an awaited call to WhenAny returns the task in the collection of tasks that finishes its download first. That task is removed from the collection and processed. The loop repeats until the collection contains no more tasks.

Prerequisites

You can follow this tutorial by using one of the following options:

- Visual Studio 2022 with the **.NET desktop development** workload installed. The .NET SDK is automatically installed when you select this workload.
- The .NET SDK with a code editor of your choice, such as Visual Studio Code.

Create example application

Create a new .NET Core console application. You can create one by using the dotnet new console command or from Visual Studio.

Open the *Program.cs* file in your code editor, and replace the existing code with this code:

C#

```
using System.Diagnostics;

namespace ProcessTasksAsTheyFinish;

class Program
{
    static void Main(string[] args)
    {
        Console.WriteLine("Hello World!");
    }
}
```

Add fields

In the `Program` class definition, add the following two fields:

C#

```
static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/maui"
};
```

The `HttpClient` exposes the ability to send HTTP requests and receive HTTP responses. The `s_urlList` holds all of the URLs that the application plans to process.

Update application entry point

The main entry point into the console application is the `Main` method. Replace the existing method with the following:

C#

```
static Task Main() => SumPageSizesAsync();
```

The updated `Main` method is now considered an Async main, which allows for an asynchronous entry point into the executable. It is expressed as a call to `SumPageSizesAsync`.

Create the asynchronous sum page sizes method

Below the `Main` method, add the `SumPageSizesAsync` method:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

The `while` loop removes one of the tasks in each iteration. After every task has completed, the loop ends. The method starts by instantiating and starting a Stopwatch. It then includes a query that, when executed, creates a collection of tasks. Each call to `ProcessUrlAsync` in the following code returns a Task<TResult>, where `TResult` is an integer:

C#

```
IEnumerable<Task<int>> downloadTasksQuery =
    from url in s_urlList
    select ProcessUrlAsync(url, s_client);
```

Due to deferred execution with the LINQ, you call Enumerable.ToList to start each task.

C#

```
List<Task<int>> downloadTasks = downloadTasksQuery.ToList();
```

The `while` loop performs the following steps for each task in the collection:

1. Awaits a call to `WhenAny` to identify the first task in the collection that has finished its download.

   C#

   ```
   Task<int> finishedTask = await Task.WhenAny(downloadTasks);
   ```

2. Removes that task from the collection.

   C#

   ```
   downloadTasks.Remove(finishedTask);
   ```

3. Awaits `finishedTask`, which is returned by a call to `ProcessUrlAsync`. The `finishedTask` variable is a Task<TResult> where `TResult` is an integer. The task is already complete, but you await it to retrieve the length of the downloaded website, as the following example shows. If the task is faulted, `await` will throw the first child exception stored in the `AggregateException`, unlike reading the Task<TResult>.Result property, which would throw the `AggregateException`.

   C#

   ```
   total += await finishedTask;
   ```

Add process method

Add the following `ProcessUrlAsync` method below the `SumPageSizesAsync` method:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

For any given URL, the method will use the `client` instance provided to get the response as a `byte[]`. The length is returned after the URL and length is written to the console.

Run the program several times to verify that the downloaded lengths don't always appear in the same order.

> ⊗ **Caution**
>
> You can use `WhenAny` in a loop, as described in the example, to solve problems that involve a small number of tasks. However, other approaches are more efficient if you have a large number of tasks to process. For more information and examples, see **Processing tasks as they complete**.

Simplify the approach using `Task.WhenEach`

The `while` loop implemented in `SumPageSizesAsync` method can be simplified using the new Task.WhenEach method introduced in .NET 9, by calling it in `await foreach` loop.
Replace the previously implemented `while` loop:

C#

```
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
        total += await finishedTask;
    }
```

with the simplified `await foreach`:

C#

```
    await foreach (Task<int> t in Task.WhenEach(downloadTasks))
    {
        total += await t;
    }
```

This new approach allows to no longer repeatedly call `Task.WhenAny` to manually call a task and remove the one that finishes, because `Task.WhenEach` iterates through task *in an order of their completion*.

Complete example

The following code is the complete text of the *Program.cs* file for the example.

C#

```
using System.Diagnostics;

HttpClient s_client = new()
{
    MaxResponseContentBufferSize = 1_000_000
};

IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/maui"
};

await SumPageSizesAsync();

async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    IEnumerable<Task<int>> downloadTasksQuery =
        from url in s_urlList
        select ProcessUrlAsync(url, s_client);

    List<Task<int>> downloadTasks = downloadTasksQuery.ToList();

    int total = 0;
    while (downloadTasks.Any())
    {
        Task<int> finishedTask = await Task.WhenAny(downloadTasks);
        downloadTasks.Remove(finishedTask);
```

```
        total += await finishedTask;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:    {total:#,#}");
    Console.WriteLine($"Elapsed time:              {stopwatch.Elapsed}\n");
}

static async Task<int> ProcessUrlAsync(string url, HttpClient client)
{
    byte[] content = await client.GetByteArrayAsync(url);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}

// Example output:
// https://learn.microsoft.com                                   132,517
// https://learn.microsoft.com/powershell                         57,375
// https://learn.microsoft.com/gaming                             33,549
// https://learn.microsoft.com/aspnet/core                        88,714
// https://learn.microsoft.com/surface                            39,840
// https://learn.microsoft.com/enterprise-mobility-security       30,903
// https://learn.microsoft.com/microsoft-365                      67,867
// https://learn.microsoft.com/windows                            26,816
// https://learn.microsoft.com/maui                               57,958
// https://learn.microsoft.com/dotnet                             78,706
// https://learn.microsoft.com/graph                              48,277
// https://learn.microsoft.com/dynamics365                        49,042
// https://learn.microsoft.com/office                             67,867
// https://learn.microsoft.com/system-center                      42,887
// https://learn.microsoft.com/education                          38,636
// https://learn.microsoft.com/azure                             421,663
// https://learn.microsoft.com/visualstudio                       30,925
// https://learn.microsoft.com/sql                                54,608
// https://learn.microsoft.com/azure/devops                       86,034

// Total bytes returned:    1,454,184
// Elapsed time:            00:00:01.1290403
```

See also

- WhenAny
- WhenEach
- Asynchronous programming with async and await (C#)

Asynchronous file access (C#)

By using the async feature to access files, you can call into asynchronous methods without using callbacks or splitting your code across multiple methods or lambda expressions. To make synchronous code asynchronous, call an asynchronous method instead of a synchronous method and add a few keywords to the code.

Consider adding asynchrony to file access calls for these reasons:

- ✓ Asynchrony makes UI applications more responsive because the UI thread that launches the operation can perform other work. If the UI thread must execute code that takes a long time (for example, more than 50 milliseconds), the UI might freeze until the I/O is complete and the UI thread can again process keyboard and mouse input and other events.
- ✓ Asynchrony improves the scalability of ASP.NET and other server-based applications by reducing the need for threads. If the application uses a dedicated thread per response and a thousand requests are being handled simultaneously, a thousand threads are needed. Asynchronous operations often don't need to use a thread during the wait. They use the existing I/O completion thread briefly at the end.
- ✓ The latency of a file access operation might be very low under current conditions, but the latency might greatly increase in the future. For example, a file might be moved to a server that's across the world.
- ✓ The added overhead of using the Async feature is small.
- ✓ Multiple asynchronous I/O operations can run without blocking the calling thread.

Use appropriate classes

The simple examples in this topic demonstrate File.WriteAllTextAsync and File.ReadAllTextAsync. For fine control over the file I/O operations, use the FileStream class, which has an option that causes asynchronous I/O to occur at the operating system level. By using this option, you can avoid blocking a thread pool thread in many cases. To enable this option, specify the `useAsync=true` or `options=FileOptions.Asynchronous` argument in the constructor call.

You can't use this option with StreamReader and StreamWriter if you open them directly by specifying a file path. However, you can use this option if you provide them a Stream that the FileStream class opened. Asynchronous calls are faster in UI apps even if a thread pool thread is blocked, because the UI thread isn't blocked during the wait.

Write text

The following examples write text to a file. At each await statement, the method immediately exits. When the file I/O is complete, the method resumes at the statement that follows the await statement. The async modifier is in the definition of methods that use the await statement.

Simple example

C#

```
public async Task SimpleWriteAsync()
{
    string filePath = "simple.txt";
    string text = $"Hello World";

    await File.WriteAllTextAsync(filePath, text);
}
```

Finite control example

C#

```
public async Task ProcessWriteAsync()
{
    string filePath = "temp.txt";
    string text = $"Hello World{Environment.NewLine}";

    await WriteTextAsync(filePath, text);
}

async Task WriteTextAsync(string filePath, string text)
{
    byte[] encodedText = Encoding.Unicode.GetBytes(text);

    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Create, FileAccess.Write, FileShare.None,
            bufferSize: 4096, useAsync: true);

    await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
}
```

The original example has the statement `await sourceStream.WriteAsync(encodedText, 0, encodedText.Length);`, which is a contraction of the following two statements:

C#

```
Task theTask = sourceStream.WriteAsync(encodedText, 0, encodedText.Length);
await theTask;
```

The first statement returns a task and causes file processing to start. The second statement with the await causes the method to immediately exit and return a different task. When the file processing later completes, execution returns to the statement that follows the await.

Read text

The following examples read text from a file.

Simple example

C#

```
public async Task SimpleReadAsync()
{
    string filePath = "simple.txt";
    string text = await File.ReadAllTextAsync(filePath);

    Console.WriteLine(text);
}
```

Finite control example

The text is buffered and, in this case, placed into a StringBuilder. Unlike in the previous example, the evaluation of the await produces a value. The ReadAsync method returns a Task<Int32>, so the evaluation of the await produces an `Int32` value `numRead` after the operation completes. For more information, see Async Return Types (C#).

C#

```
public async Task ProcessReadAsync()
{
    try
    {
        string filePath = "temp.txt";
        if (File.Exists(filePath) != false)
        {
            string text = await ReadTextAsync(filePath);
            Console.WriteLine(text);
        }
        else
        {
            Console.WriteLine($"file not found: {filePath}");
        }
```

```
    }
    catch (Exception ex)
    {
        Console.WriteLine(ex.Message);
    }
}

async Task<string> ReadTextAsync(string filePath)
{
    using var sourceStream =
        new FileStream(
            filePath,
            FileMode.Open, FileAccess.Read, FileShare.Read,
            bufferSize: 4096, useAsync: true);

    var sb = new StringBuilder();

    byte[] buffer = new byte[0x1000];
    int numRead;
    while ((numRead = await sourceStream.ReadAsync(buffer, 0, buffer.Length)) != 0)
    {
        string text = Encoding.Unicode.GetString(buffer, 0, numRead);
        sb.Append(text);
    }

    return sb.ToString();
}
```

Multiple asynchronous I/O operations

The following examples start multiple async write operations. The runtime queues these operations, and the underlying implementation might use operating system (OS) async I/O or thread pool threads depending on the platform and configuration, so actual concurrency depends on OS and hardware.

Simple example

C#

```
public async Task SimpleParallelWriteAsync()
{
    string folder = Directory.CreateDirectory("tempfolder").Name;
    IList<Task> writeTaskList = new List<Task>();

    for (int index = 11; index <= 20; ++ index)
    {
        string fileName = $"file-{index:00}.txt";
        string filePath = $"{folder}/{fileName}";
        string text = $"In file {index}{Environment.NewLine}";
```

```
        writeTaskList.Add(File.WriteAllTextAsync(filePath, text));
    }

    await Task.WhenAll(writeTaskList);
}
```

Finite control example

For each file, the WriteAsync method returns a task that's added to a list of tasks. The `await Task.WhenAll(tasks);` statement exits the method and resumes within the method when file processing is complete for all of the tasks.

The example closes all FileStream instances in a `finally` block after the tasks are complete. If each `FileStream` was instead created in a `using` statement, the `FileStream` might be disposed of before the task was complete.

The async approach avoids blocking the calling thread while I/O is pending. In many cases, throughput improvements depend on the OS, the hardware, and, on some platforms, .NET runtime behavior such as thread pool limits and scheduling.

C#

```
public async Task ProcessMultipleWritesAsync()
{
    IList<FileStream> sourceStreams = new List<FileStream>();

    try
    {
        string folder = Directory.CreateDirectory("tempfolder").Name;
        IList<Task> writeTaskList = new List<Task>();

        for (int index = 1; index <= 10; ++ index)
        {
            string fileName = $"file-{index:00}.txt";
            string filePath = $"{folder}/{fileName}";

            string text = $"In file {index}{Environment.NewLine}";
            byte[] encodedText = Encoding.Unicode.GetBytes(text);

            var sourceStream =
                new FileStream(
                    filePath,
                    FileMode.Create, FileAccess.Write, FileShare.None,
                    bufferSize: 4096, useAsync: true);

            Task writeTask = sourceStream.WriteAsync(encodedText, 0,
encodedText.Length);
            sourceStreams.Add(sourceStream);
```

```
            writeTaskList.Add(writeTask);
        }

        await Task.WhenAll(writeTaskList);
    }
    finally
    {
        foreach (FileStream sourceStream in sourceStreams)
        {
            sourceStream.Close();
        }
    }
}
```

When using the WriteAsync and ReadAsync methods, you can specify a CancellationToken to cancel the operation mid-stream. For more information, see Cancellation in managed threads.

See also

- Asynchronous programming with async and await (C#)
- Async return types (C#)

ⓘ **Note:** The author created this article with assistance from AI. Learn more

---

Last updated on 04/23/2026

Cancel a list of tasks

Article • 03/19/2025

You can cancel an async console application if you don't want to wait for it to finish. By following the example in this topic, you can add a cancellation to an application that downloads the contents of a list of websites. You can cancel many tasks by associating the CancellationTokenSource instance with each task. If you select the `Enter` key, you cancel all tasks that aren't yet complete.

This tutorial covers:

- ✓ Creating a .NET console application
- ✓ Writing an async application that supports cancellation
- ✓ Demonstrating signaling cancellation

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Create example application

Create a new .NET Core console application. You can create one by using the dotnet new console command or from Visual Studio. Open the *Program.cs* file in your favorite code editor.

Replace using directives

Replace the existing `using` directives with these declarations:

C#

```
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Net.Http;
using System.Threading;
using System.Threading.Tasks;
```

Add fields

In the `Program` class definition, add these three fields:

C#

```
static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

static readonly HttpClient s_client = new HttpClient
{
    MaxResponseContentBufferSize = 1_000_000
};

static readonly IEnumerable<string> s_urlList = new string[]
{
    "https://learn.microsoft.com",
    "https://learn.microsoft.com/aspnet/core",
    "https://learn.microsoft.com/azure",
    "https://learn.microsoft.com/azure/devops",
    "https://learn.microsoft.com/dotnet",
    "https://learn.microsoft.com/dynamics365",
    "https://learn.microsoft.com/education",
    "https://learn.microsoft.com/enterprise-mobility-security",
    "https://learn.microsoft.com/gaming",
    "https://learn.microsoft.com/graph",
    "https://learn.microsoft.com/microsoft-365",
    "https://learn.microsoft.com/office",
    "https://learn.microsoft.com/powershell",
    "https://learn.microsoft.com/sql",
    "https://learn.microsoft.com/surface",
    "https://learn.microsoft.com/system-center",
    "https://learn.microsoft.com/visualstudio",
    "https://learn.microsoft.com/windows",
    "https://learn.microsoft.com/maui"
};
```

The CancellationTokenSource is used to signal a requested cancellation to a CancellationToken. The `HttpClient` exposes the ability to send HTTP requests and receive HTTP responses. The `s_urlList` holds all of the URLs that the application plans to process.

Update application entry point

The main entry point into the console application is the `Main` method. Replace the existing method with the following:

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");
    Console.WriteLine("Press the ENTER key to cancel...\n");

    Task cancelTask = Task.Run(() =>
    {
        while (Console.ReadKey().Key != ConsoleKey.Enter)
        {
            Console.WriteLine("Press the ENTER key to cancel...");
        }

        Console.WriteLine("\nENTER key pressed: cancelling downloads.\n");
        s_cts.Cancel();
    });

    Task sumPageSizesTask = SumPageSizesAsync();

    Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
    if (finishedTask == cancelTask)
    {
        // wait for the cancellation to take place:
        try
        {
            await sumPageSizesTask;
            Console.WriteLine("Download task completed before cancel request
was processed.");
        }
        catch (TaskCanceledException)
        {
            Console.WriteLine("Download task has been cancelled.");
        }
    }

    Console.WriteLine("Application ending.");
}
```

The updated `Main` method is now considered an Async main, which allows for an asynchronous entry point into the executable. It writes a few instructional messages to the console, then declares a Task instance named `cancelTask`, which will read console key strokes. If the `Enter` key is pressed, a call to CancellationTokenSource.Cancel() is made. This will signal cancellation. Next, the `sumPageSizesTask` variable is assigned from the `SumPageSizesAsync` method. Both tasks are then passed to Task.WhenAny(Task[]), which will continue when any of the two tasks have completed.

The next block of code ensures that the application doesn't exit until the cancellation has been processed. If the first task to complete is the `cancelTask`, the `sumPageSizeTask` is awaited. If it was cancelled, when awaited it throws a

System.Threading.Tasks.TaskCanceledException. The block catches that exception, and prints a message.

Create the asynchronous sum page sizes method

Below the `Main` method, add the `SumPageSizesAsync` method:

C#

```
static async Task SumPageSizesAsync()
{
    var stopwatch = Stopwatch.StartNew();

    int total = 0;
    foreach (string url in s_urlList)
    {
        int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
        total += contentLength;
    }

    stopwatch.Stop();

    Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
    Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
}
```

The method starts by instantiating and starting a Stopwatch. It then loops through each URL in the `s_urlList` and calls `ProcessUrlAsync`. With each iteration, the `s_cts.Token` is passed into the `ProcessUrlAsync` method and the code returns a Task<TResult>, where `TResult` is an integer:

C#

```
int total = 0;
foreach (string url in s_urlList)
{
    int contentLength = await ProcessUrlAsync(url, s_client, s_cts.Token);
    total += contentLength;
}
```

Add process method

Add the following `ProcessUrlAsync` method below the `SumPageSizesAsync` method:

C#

```
static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
{
    HttpResponseMessage response = await client.GetAsync(url, token);
    byte[] content = await response.Content.ReadAsByteArrayAsync(token);
    Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

    return content.Length;
}
```

For any given URL, the method will use the `client` instance provided to get the response as a `byte[]`. The CancellationToken instance is passed into the HttpClient.GetAsync(String, CancellationToken) and HttpContent.ReadAsByteArrayAsync() methods. The `token` is used to register for requested cancellation. The length is returned after the URL and length is written to the console.

Example application output

Console

```
Application started.
Press the ENTER key to cancel...

https://learn.microsoft.com                                   37,357
https://learn.microsoft.com/aspnet/core                       85,589
https://learn.microsoft.com/azure                            398,939
https://learn.microsoft.com/azure/devops                      73,663
https://learn.microsoft.com/dotnet                            67,452
https://learn.microsoft.com/dynamics365                       48,582
https://learn.microsoft.com/education                         22,924

ENTER key pressed: cancelling downloads.

Application ending.
```

Complete example

The following code is the complete text of the *Program.cs* file for the example.

C#

```
using System.Diagnostics;
```

```
class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();

    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/maui"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");
        Console.WriteLine("Press the ENTER key to cancel...\n");

        Task cancelTask = Task.Run(() =>
        {
            while (Console.ReadKey().Key != ConsoleKey.Enter)
            {
                Console.WriteLine("Press the ENTER key to cancel...");
            }

            Console.WriteLine("\nENTER key pressed: cancelling
downloads.\n");
            s_cts.Cancel();
        });

        Task sumPageSizesTask = SumPageSizesAsync();

        Task finishedTask = await Task.WhenAny(new[] { cancelTask,
sumPageSizesTask });
        if (finishedTask == cancelTask)
```

```
        {
            // wait for the cancellation to take place:
            try
            {
                await sumPageSizesTask;
                Console.WriteLine("Download task completed before cancel
request was processed.");
            }
            catch (OperationCanceledException)
            {
                Console.WriteLine("Download task has been cancelled.");
            }
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

See also

- CancellationToken
- CancellationTokenSource
- Asynchronous programming with async and await (C#)

Next steps

Cancel async tasks after a period of time (C#)

Cancel async tasks after a period of time

Article • 09/08/2023

You can cancel an asynchronous operation after a period of time by using the CancellationTokenSource.CancelAfter method if you don't want to wait for the operation to finish. This method schedules the cancellation of any associated tasks that aren't complete within the period of time that's designated by the `CancelAfter` expression.

This example adds to the code that's developed in Cancel a list of tasks (C#) to download a list of websites and to display the length of the contents of each one.

This tutorial covers:

- ✓ Updating an existing .NET console application
- ✓ Scheduling a cancellation

Prerequisites

This tutorial requires the following:

- You're expected to have created an application in the Cancel a list of tasks (C#) tutorial
- .NET 5 or later SDK
- Integrated development environment (IDE)
  - We recommend Visual Studio or Visual Studio Code

Update application entry point

Replace the existing `Main` method with the following:

C#

```
static async Task Main()
{
    Console.WriteLine("Application started.");

    try
    {
        s_cts.CancelAfter(3500);

        await SumPageSizesAsync();
    }
    catch (OperationCanceledException)
    {
```

```
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }

        Console.WriteLine("Application ending.");
    }
```

The updated `Main` method writes a few instructional messages to the console. Within the try-catch, a call to CancellationTokenSource.CancelAfter(Int32) schedules a cancellation. This will signal cancellation after a period of time.

Next, the `SumPageSizesAsync` method is awaited. If processing all of the URLs occurs faster than the scheduled cancellation, the application ends. However, if the scheduled cancellation is triggered before all of the URLs are processed, a OperationCanceledException is thrown.

Example application output

Console

```
Application started.

https://learn.microsoft.com                                37,357
https://learn.microsoft.com/aspnet/core                    85,589
https://learn.microsoft.com/azure                         398,939
https://learn.microsoft.com/azure/devops                   73,663

Tasks cancelled: timed out.

Application ending.
```

Complete example

The following code is the complete text of the *Program.cs* file for the example.

C#

```
using System.Diagnostics;

class Program
{
    static readonly CancellationTokenSource s_cts = new
CancellationTokenSource();
```

```
    static readonly HttpClient s_client = new HttpClient
    {
        MaxResponseContentBufferSize = 1_000_000
    };

    static readonly IEnumerable<string> s_urlList = new string[]
    {
            "https://learn.microsoft.com",
            "https://learn.microsoft.com/aspnet/core",
            "https://learn.microsoft.com/azure",
            "https://learn.microsoft.com/azure/devops",
            "https://learn.microsoft.com/dotnet",
            "https://learn.microsoft.com/dynamics365",
            "https://learn.microsoft.com/education",
            "https://learn.microsoft.com/enterprise-mobility-security",
            "https://learn.microsoft.com/gaming",
            "https://learn.microsoft.com/graph",
            "https://learn.microsoft.com/microsoft-365",
            "https://learn.microsoft.com/office",
            "https://learn.microsoft.com/powershell",
            "https://learn.microsoft.com/sql",
            "https://learn.microsoft.com/surface",
            "https://learn.microsoft.com/system-center",
            "https://learn.microsoft.com/visualstudio",
            "https://learn.microsoft.com/windows",
            "https://learn.microsoft.com/maui"
    };

    static async Task Main()
    {
        Console.WriteLine("Application started.");

        try
        {
            s_cts.CancelAfter(3500);

            await SumPageSizesAsync();
        }
        catch (OperationCanceledException)
        {
            Console.WriteLine("\nTasks cancelled: timed out.\n");
        }
        finally
        {
            s_cts.Dispose();
        }

        Console.WriteLine("Application ending.");
    }

    static async Task SumPageSizesAsync()
    {
        var stopwatch = Stopwatch.StartNew();

        int total = 0;
```

```
        foreach (string url in s_urlList)
        {
            int contentLength = await ProcessUrlAsync(url, s_client,
s_cts.Token);
            total += contentLength;
        }

        stopwatch.Stop();

        Console.WriteLine($"\nTotal bytes returned:  {total:#,#}");
        Console.WriteLine($"Elapsed time:          {stopwatch.Elapsed}\n");
    }

    static async Task<int> ProcessUrlAsync(string url, HttpClient client,
CancellationToken token)
    {
        HttpResponseMessage response = await client.GetAsync(url, token);
        byte[] content = await response.Content.ReadAsByteArrayAsync(token);
        Console.WriteLine($"{url,-60} {content.Length,10:#,#}");

        return content.Length;
    }
}
```

See also

- CancellationToken
- CancellationTokenSource
- Asynchronous programming with async and await (C#)
- Cancel a list of tasks (C#)

Tutorial: Generate and consume async streams using C# and .NET

Article • 03/25/2023

Async streams model a streaming source of data. Data streams often retrieve or generate elements asynchronously. They provide a natural programming model for asynchronous streaming data sources.

In this tutorial, you'll learn how to:

- ✓ Create a data source that generates a sequence of data elements asynchronously.
- ✓ Consume that data source asynchronously.
- ✓ Support cancellation and captured contexts for asynchronous streams.
- ✓ Recognize when the new interface and data source are preferred to earlier synchronous data sequences.

Prerequisites

You'll need to set up your machine to run .NET, including the C# compiler. The C# compiler is available with Visual Studio 2022 or the .NET SDK.

You'll need to create a GitHub access token so that you can access the GitHub GraphQL endpoint. Select the following permissions for your GitHub Access Token:

- repo:status
- public_repo

Save the access token in a safe place so you can use it to gain access to the GitHub API endpoint.

> ⚠ **Warning**
>
> Keep your personal access token secure. Any software with your personal access token could make GitHub API calls using your access rights.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or the .NET CLI.

Run the starter application

You can get the code for the starter application used in this tutorial from the dotnet/docs repository in the asynchronous-programming/snippets folder.

The starter application is a console application that uses the GitHub GraphQL interface to retrieve recent issues written in the dotnet/docs repository. Start by looking at the following code for the starter app `Main` method:

C#

```
static async Task Main(string[] args)
{
    //Follow these steps to create a GitHub Access Token
    // https://help.github.com/articles/creating-a-personal-access-token-
for-the-command-line/#creating-a-token
    //Select the following permissions for your GitHub Access Token:
    // - repo:status
    // - public_repo
    // Replace the 3rd parameter to the following code with your GitHub
access token.
    var key = GetEnvVariable("GitHubKey",
    "You must store your GitHub key in the 'GitHubKey' environment
variable",
    "");

    var client = new GitHubClient(new
Octokit.ProductHeaderValue("IssueQueryDemo"))
    {
        Credentials = new Octokit.Credentials(key)
    };

    var progressReporter = new progressStatus((num) =>
    {
        Console.WriteLine($"Received {num} issues in total");
    });
    CancellationTokenSource cancellationSource = new
CancellationTokenSource();

    try
    {
        var results = await RunPagedQueryAsync(client, PagedIssueQuery,
"docs",
            cancellationSource.Token, progressReporter);
        foreach(var issue in results)
            Console.WriteLine(issue);
    }
    catch (OperationCanceledException)
    {
        Console.WriteLine("Work has been cancelled");
    }
}
```

You can either set a `GitHubKey` environment variable to your personal access token, or you can replace the last argument in the call to `GetEnvVariable` with your personal access token. Don't put your access code in source code if you'll be sharing the source with others. Never upload access codes to a shared source repository.

After creating the GitHub client, the code in `Main` creates a progress reporting object and a cancellation token. Once those objects are created, `Main` calls `RunPagedQueryAsync` to retrieve the most recent 250 created issues. After that task has finished, the results are displayed.

When you run the starter application, you can make some important observations about how this application runs. You'll see progress reported for each page returned from GitHub. You can observe a noticeable pause before GitHub returns each new page of issues. Finally, the issues are displayed only after all 10 pages have been retrieved from GitHub.

Examine the implementation

The implementation reveals why you observed the behavior discussed in the previous section. Examine the code for `RunPagedQueryAsync`:

C#

```
private static async Task<JArray> RunPagedQueryAsync(GitHubClient client,
string queryText, string repoName, CancellationToken cancel, IProgress<int>
progress)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    JArray finalResults = new JArray();
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
```

```
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();
        finalResults.Merge(issues(results)["nodes"]!);
        progress?.Report(issuesReturned);
        cancel.ThrowIfCancellationRequested();
    }
    return finalResults;

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

The very first thing this method does is to create the POST object, using the `GraphQLRequest` class:

C#

```
public class GraphQLRequest
{
    [JsonProperty("query")]
    public string? Query { get; set; }

    [JsonProperty("variables")]
    public IDictionary<string, object> Variables { get; } = new
Dictionary<string, object>();

    public string ToJsonText() =>
        JsonConvert.SerializeObject(this);
}
```

which helps to form the POST object body, and correctly convert it to JSON presented as single string with the `ToJsonText` method, which removes all newline characters from your request body marking them with the `\` (backslash) escape character.

Let's concentrate on the paging algorithm and async structure of the preceding code. (You can consult the GitHub GraphQL documentation for details on the GitHub GraphQL API.) The `RunPagedQueryAsync` method enumerates the issues from most recent to oldest. It requests 25 issues per page and examines the `pageInfo` structure of the response to continue with the previous page. That follows GraphQL's standard paging support for multi-page responses. The response includes a `pageInfo` object that includes a `hasPreviousPages` value and a `startCursor` value used to request the

previous page. The issues are in the `nodes` array. The `RunPagedQueryAsync` method appends these nodes to an array that contains all the results from all pages.

After retrieving and restoring a page of results, `RunPagedQueryAsync` reports progress and checks for cancellation. If cancellation has been requested, `RunPagedQueryAsync` throws an OperationCanceledException.

There are several elements in this code that can be improved. Most importantly, `RunPagedQueryAsync` must allocate storage for all the issues returned. This sample stops at 250 issues because retrieving all open issues would require much more memory to store all the retrieved issues. The protocols for supporting progress reports and cancellation make the algorithm harder to understand on its first reading. More types and APIs are involved. You must trace the communications through the CancellationTokenSource and its associated CancellationToken to understand where cancellation is requested and where it's granted.

Async streams provide a better way

Async streams and the associated language support address all those concerns. The code that generates the sequence can now use `yield return` to return elements in a method that was declared with the `async` modifier. You can consume an async stream using an `await foreach` loop just as you consume any sequence using a `foreach` loop.

These new language features depend on three new interfaces added to .NET Standard 2.1 and implemented in .NET Core 3.0:

- System.Collections.Generic.IAsyncEnumerable<T>
- System.Collections.Generic.IAsyncEnumerator<T>
- System.IAsyncDisposable

These three interfaces should be familiar to most C# developers. They behave in a manner similar to their synchronous counterparts:

- System.Collections.Generic.IEnumerable<T>
- System.Collections.Generic.IEnumerator<T>
- System.IDisposable

One type that may be unfamiliar is System.Threading.Tasks.ValueTask. The `ValueTask` struct provides a similar API to the System.Threading.Tasks.Task class. `ValueTask` is used in these interfaces for performance reasons.

Convert to async streams

Next, convert the `RunPagedQueryAsync` method to generate an async stream. First, change the signature of `RunPagedQueryAsync` to return an `IAsyncEnumerable<JToken>`, and remove the cancellation token and progress objects from the parameter list as shown in the following code:

```
C#

private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
```

The starter code processes each page as the page is retrieved, as shown in the following code:

```
C#

finalResults.Merge(issues(results)["nodes"]!);
progress?.Report(issuesReturned);
cancel.ThrowIfCancellationRequested();
```

Replace those three lines with the following code:

```
C#

foreach (JObject issue in issues(results)["nodes"]!)
    yield return issue;
```

You can also remove the declaration of `finalResults` earlier in this method and the `return` statement that follows the loop you modified.

You've finished the changes to generate an async stream. The finished method should resemble the following code:

```
C#

private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;
```

```
    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

Next, you change the code that consumes the collection to consume the async stream. Find the following code in `Main` that processes the collection of issues:

C#

```
var progressReporter = new progressStatus((num) =>
{
    Console.WriteLine($"Received {num} issues in total");
});
CancellationTokenSource cancellationSource = new CancellationTokenSource();

try
{
    var results = await RunPagedQueryAsync(client, PagedIssueQuery, "docs",
        cancellationSource.Token, progressReporter);
    foreach(var issue in results)
        Console.WriteLine(issue);
}
catch (OperationCanceledException)
{
```

```
    Console.WriteLine("Work has been cancelled");
}
```

Replace that code with the following `await foreach` loop:

C#

```
int num = 0;
await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs"))
{
    Console.WriteLine(issue);
    Console.WriteLine($"Received {++num} issues in total");
}
```

The new interface IAsyncEnumerator<T> derives from IAsyncDisposable. That means the preceding loop will asynchronously dispose the stream when the loop finishes. You can imagine the loop looks like the following code:

C#

```
int num = 0;
var enumerator = RunPagedQueryAsync(client, PagedIssueQuery,
"docs").GetAsyncEnumerator();
try
{
    while (await enumerator.MoveNextAsync())
    {
        var issue = enumerator.Current;
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
} finally
{
    if (enumerator != null)
        await enumerator.DisposeAsync();
}
```

By default, stream elements are processed in the captured context. If you want to disable capturing of the context, use the TaskAsyncEnumerableExtensions.ConfigureAwait extension method. For more information about synchronization contexts and capturing the current context, see the article on consuming the Task-based asynchronous pattern.

Async streams support cancellation using the same protocol as other `async` methods. You would modify the signature for the async iterator method as follows to support cancellation:

C#

```
private static async IAsyncEnumerable<JToken>
RunPagedQueryAsync(GitHubClient client,
    string queryText, string repoName, [EnumeratorCancellation]
CancellationToken cancellationToken = default)
{
    var issueAndPRQuery = new GraphQLRequest
    {
        Query = queryText
    };
    issueAndPRQuery.Variables["repo_name"] = repoName;

    bool hasMorePages = true;
    int pagesReturned = 0;
    int issuesReturned = 0;

    // Stop with 10 pages, because these are large repos:
    while (hasMorePages && (pagesReturned++ < 10))
    {
        var postBody = issueAndPRQuery.ToJsonText();
        var response = await client.Connection.Post<string>(new
Uri("https://api.github.com/graphql"),
            postBody, "application/json", "application/json");

        JObject results =
JObject.Parse(response.HttpResponse.Body.ToString()!);

        int totalCount = (int)issues(results)["totalCount"]!;
        hasMorePages = (bool)pageInfo(results)["hasPreviousPage"]!;
        issueAndPRQuery.Variables["start_cursor"] = pageInfo(results)
["startCursor"]!.ToString();
        issuesReturned += issues(results)["nodes"]!.Count();

        foreach (JObject issue in issues(results)["nodes"]!)
            yield return issue;
    }

    JObject issues(JObject result) => (JObject)result["data"]!
["repository"]!["issues"]!;
    JObject pageInfo(JObject result) => (JObject)issues(result)
["pageInfo"]!;
}
```

The System.Runtime.CompilerServices.EnumeratorCancellationAttribute attribute causes the compiler to generate code for the IAsyncEnumerator<T> that makes the token passed to `GetAsyncEnumerator` visible to the body of the async iterator as that argument. Inside `runQueryAsync`, you could examine the state of the token and cancel further work if requested.

You use another extension method, WithCancellation, to pass the cancellation token to the async stream. You would modify the loop enumerating the issues as follows:

```
C#
```

```
private static async Task EnumerateWithCancellation(GitHubClient client)
{
    int num = 0;
    var cancellation = new CancellationTokenSource();
    await foreach (var issue in RunPagedQueryAsync(client, PagedIssueQuery,
"docs")
        .WithCancellation(cancellation.Token))
    {
        Console.WriteLine(issue);
        Console.WriteLine($"Received {++num} issues in total");
    }
}
```

You can get the code for the finished tutorial from the dotnet/docs repository in the asynchronous-programming/snippets folder.

Run the finished application

Run the application again. Contrast its behavior with the behavior of the starter application. The first page of results is enumerated as soon as it's available. There's an observable pause as each new page is requested and retrieved, then the next page's results are quickly enumerated. The `try` / `catch` block isn't needed to handle cancellation: the caller can stop enumerating the collection. Progress is clearly reported because the async stream generates results as each page is downloaded. The status for each issue returned is seamlessly included in the `await foreach` loop. You don't need a callback object to track progress.

You can see improvements in memory use by examining the code. You no longer need to allocate a collection to store all the results before they're enumerated. The caller can determine how to consume the results and if a storage collection is needed.

Run both the starter and finished applications and you can observe the differences between the implementations for yourself. You can delete the GitHub access token you created when you started this tutorial after you've finished. If an attacker gained access to that token, they could access GitHub APIs using your credentials.

In this tutorial, you used async streams to read a individual items from a network API that returns pages of data. Async streams can also read from "never ending streams"

like a stock ticker, or sensor device. The call to `MoveNextAsync` returns the next item as soon as it's available.

Nullable reference types

Article • 12/14/2024

Nullable reference types are a group of features that minimize the likelihood that your code causes the runtime to throw System.NullReferenceException. Three features that help you avoid these exceptions, including the ability to explicitly mark a reference type as *nullable*:

- Improved static flow analysis that determines if a variable might be `null` before dereferencing it.
- Attributes that annotate APIs so that the flow analysis determines *null-state*.
- Variable annotations that developers use to explicitly declare the intended *null-state* for a variable.

The compiler tracks the *null-state* of every expression in your code at compile time. The *null-state* has one of two values:

- *not-null*: The expression is known to be not-`null`.
- *maybe-null*: The expression might be `null`.

Variable annotations determine the *nullability* of a reference type variable:

- *non-nullable*: If you assign a `null` value or a *maybe-null* expression to the variable, the compiler issues a warning. Variables that are *non-nullable* have a default null-state of *not-null*.
- *nullable*: You can assign a `null` value or a *maybe-null* expression to the variable. When the variable's null-state is *maybe-null*, the compiler issues a warning if you dereference the variable. The default null-state for the variable is *maybe-null*.

The rest of this article describes how those three feature areas work to produce warnings when your code might **dereference** a `null` value. Dereferencing a variable means to access one of its members using the `.` (dot) operator, as shown in the following example:

C#

```
string message = "Hello, World!";
int length = message.Length; // dereferencing "message"
```

When you dereference a variable whose value is `null`, the runtime throws a System.NullReferenceException.

Similarly warnings can be produced when `[]` notation is used to access a member of an object when the object is `null`:

```
C#
```

```
using System;

public class Collection<T>
{
    private T[] array = new T[100];
    public T this[int index]
    {
        get => array[index];
        set => array[index] = value;
    }
}

public static void Main()
{
    Collection<int> c = default;
    c[10] = 1;    // CS8602: Possible dereference of null
}
```

You'll learn about:

- The compiler's null-state analysis: how the compiler determines if an expression is not-null, or maybe-null.
- Attributes that are applied to APIs that provide more context for the compiler's null-state analysis.
- Nullable variable annotations that provide information about your intent for variables. Annotations are useful for fields, parameters, and return values to set the default null-state.
- The rules governing generic type arguments. New constraints were added because type parameters can be reference types or value types. The `?` suffix is implemented differently for nullable value types and nullable reference types.
- The Nullable context help you migrate large projects. You can enable warnings and annotations in the nullable context in parts of your app as you migrate. After you address more warnings, you can enable both settings for the entire project.

Finally, you learn known pitfalls for null-state analysis in `struct` types and arrays.

You can also explore these concepts in our Learn module on Nullable safety in C#.

Null-state analysis

Null-state analysis tracks the *null-state* of references. An expression is either *not-null* or *maybe-null*. The compiler determines that a variable is *not-null* in two ways:

1. The variable was assigned a value that is known to be *not-null*.
2. The variable was checked against `null` and wasn't assigned since that check.

Any variable that the compiler can't determined as *not-null* is considered *maybe-null*. The analysis provides warnings in situations where you might accidentally dereference a `null` value. The compiler produces warnings based on the *null-state*.

- When a variable is *not-null*, that variable can be dereferenced safely.
- When a variable is *maybe-null*, that variable must be checked to ensure that it isn't `null` before dereferencing it.

Consider the following example:

C#

```
string? message = null;

// warning: dereference null.
Console.WriteLine($"The length of the message is {message.Length}");

var originalMessage = message;
message = "Hello, World!";

// No warning. Analysis determined "message" is not-null.
Console.WriteLine($"The length of the message is {message.Length}");

// warning!
Console.WriteLine(originalMessage.Length);
```

In the preceding example, the compiler determines that `message` is *maybe-null* when the first message is printed. There's no warning for the second message. The final line of code produces a warning because `originalMessage` might be null. The following example shows a more practical use for traversing a tree of nodes to the root, processing each node during the traversal:

C#

```
void FindRoot(Node node, Action<Node> processNode)
{
    for (var current = node; current != null; current = current.Parent)
    {
        processNode(current);
    }
}
```

The previous code doesn't generate any warnings for dereferencing the variable `current`. Static analysis determines that `current` is never dereferenced when it's *maybe-null*. The variable `current` is checked against `null` before `current.Parent` is accessed, and before passing `current` to the `ProcessNode` action. The previous examples show how the compiler determines *null-state* for local variables when initialized, assigned, or compared to `null`.

The null-state analysis doesn't trace into called methods. As a result, fields initialized in a common helper method called by all constructors might generate a warning with the following message:

> Non-nullable property '*name*' must contain a non-null value when exiting constructor.

You can address these warnings in one of two ways: *Constructor chaining*, or *nullable attributes* on the helper method. The following code shows an example of each. The `Person` class uses a common constructor called by all other constructors. The `Student` class has a helper method annotated with the System.Diagnostics.CodeAnalysis.MemberNotNullAttribute attribute:

C#

```
using System.Diagnostics.CodeAnalysis;

public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }

    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public Person() : this("John", "Doe") { }
}

public class Student : Person
{
    public string Major { get; set; }

    public Student(string firstName, string lastName, string major)
        : base(firstName, lastName)
    {
        SetMajor(major);
    }
```

```
    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
        SetMajor();
    }

    public Student()
    {
        SetMajor();
    }

    [MemberNotNull(nameof(Major))]
    private void SetMajor(string? major = default)
    {
        Major = major ?? "Undeclared";
    }
}
```

Nullable state analysis and the warnings the compiler generates help you avoid program errors by dereferencing `null`. The article on resolving nullable warnings provides techniques for correcting the warnings most likely seen in your code. The diagnostics produced from null state analysis are warnings only.

Attributes on API signatures

The null-state analysis needs hints from developers to understand the semantics of APIs. Some APIs provide null checks, and should change the *null-state* of a variable from *maybe-null* to *not-null*. Other APIs return expressions that are *not-null* or *maybe-null* depending on the *null-state* of the input arguments. For example, consider the following code that displays a message in upper case:

C#

```
void PrintMessageUpper(string? message)
{
    if (!IsNull(message))
    {
        Console.WriteLine($"{DateTime.Now}: {message.ToUpper()}");
    }
}

bool IsNull(string? s) => s == null;
```

Based on inspection, any developer would consider this code safe, and shouldn't generate warnings. However the compiler doesn't know that `IsNull` provides a null

check and issues a warning for the `message.ToUpper()` statement, considering `message` to be a *maybe-null* variable. Use the NotNullWhen attribute to fix this warning:

```
C#
```

```
bool IsNull([NotNullWhen(false)] string? s) => s == null;
```

This attribute informs the compiler, that, if `IsNull` returns `false`, the parameter `s` isn't null. The compiler changes the *null-state* of `message` to *not-null* inside the `if (!IsNull(message)) {...}` block. No warnings are issued.

Attributes provide detailed information about the null-state of arguments, return values, and members of the object instance used to invoke a member. The details on each attribute can be found in the language reference article on nullable reference attributes. As of .NET 5, all .NET runtime APIs are annotated. You improve the static analysis by annotating your APIs to provide semantic information about the *null-state* of arguments and return values.

Nullable variable annotations

The *null-state* analysis provides robust analysis for local variables. The compiler needs more information from you for member variables. The compiler needs more information to set the *null-state* of all fields at the opening bracket of a member. Any of the accessible constructors could be used to initialize the object. If a member field might ever be set to `null`, the compiler must assume its *null-state* is *maybe-null* at the start of each method.

You use annotations that can declare whether a variable is a **nullable reference type** or a **non-nullable reference type**. These annotations make important statements about the *null-state* for variables:

- **A reference isn't supposed to be null**. The default state of a non-nullable reference variable is *not-null*. The compiler enforces rules that ensure it's safe to dereference these variables without first checking that it isn't null:
  - The variable must be initialized to a non-null value.
  - The variable can never be assigned the value `null`. The compiler issues a warning when code assigns a *maybe-null* expression to a variable that shouldn't be null.
- **A reference might be null**. The default state of a nullable reference variable is *maybe-null*. The compiler enforces rules to ensure that you correctly check for a `null` reference:

- The variable can only be dereferenced when the compiler can guarantee that the value isn't `null`.
- These variables can be initialized with the default `null` value and can be assigned the value `null` in other code.
- The compiler doesn't issue warnings when code assigns a *maybe-null* expression to a variable that might be null.

Any non-nullable reference variable has the initial *null-state* of *not-null*. Any nullable reference variable has the initial *null-state* of *maybe-null*.

A **nullable reference type** is noted using the same syntax as nullable value types: a `?` is appended to the type of the variable. For example, the following variable declaration represents a nullable string variable, `name`:

```
C#

string? name;
```

When nullable reference types are enabled, any variable where the `?` isn't appended to the type name is a **non-nullable reference type**. That includes all reference type variables in existing code once you enable this feature. However, any implicitly typed local variables (declared using `var`) are **nullable reference types**. As the preceding sections showed, static analysis determines the *null-state* of local variables to determine if they're *maybe-null* before dereferencing it.

Sometimes you must override a warning when you know a variable isn't null, but the compiler determines its *null-state* is *maybe-null*. You use the null-forgiving operator `!` following a variable name to force the *null-state* to be *not-null*. For example, if you know the `name` variable isn't `null` but the compiler issues a warning, you can write the following code to override the compiler's analysis:

```
C#

name!.Length;
```

Nullable reference types and nullable value types provide a similar semantic concept: A variable can represent a value or object, or that variable might be `null`. However, nullable reference types and nullable value types are implemented differently: nullable value types are implemented using System.Nullable<T>, and nullable reference types are implemented by attributes read by the compiler. For example, `string?` and `string`

are both represented by the same type: System.String. However, `int?` and `int` are represented by `System.Nullable<System.Int32>` and System.Int32, respectively.

Nullable reference types are a compile time feature. That means it's possible for callers to ignore warnings, intentionally use `null` as an argument to a method expecting a non nullable reference. Library authors should include run-time checks against null argument values. The ArgumentNullException.ThrowIfNull is the preferred option for checking a parameter against null at run time. Furthermore, the runtime behavior of a program making use of nullable annotations is the same if all the nullable annotations, (`?` and `!`), are removed. Their only purpose is expressing design intent and providing information for null state analysis.

> **ⓘ Important**
>
> Enabling nullable annotations can change how Entity Framework Core determines if a data member is required. You can learn more details in the article on **Entity Framework Core Fundamentals: Working with Nullable Reference Types**.

Generics

Generics require detailed rules to handle `T?` for any type parameter `T`. The rules are necessarily detailed because of history and the different implementation for a nullable value type and a nullable reference type. Nullable value types are implemented using the System.Nullable<T> struct. Nullable reference types are implemented as type annotations that provide semantic rules to the compiler.

- If the type argument for `T` is a reference type, `T?` references the corresponding nullable reference type. For example, if `T` is a `string`, then `T?` is a `string?`.
- If the type argument for `T` is a value type, `T?` references the same value type, `T`. For example, if `T` is an `int`, the `T?` is also an `int`.
- If the type argument for `T` is a nullable reference type, `T?` references that same nullable reference type. For example, if `T` is a `string?`, then `T?` is also a `string?`.
- If the type argument for `T` is a nullable value type, `T?` references that same nullable value type. For example, if `T` is a `int?`, then `T?` is also a `int?`.

For return values, `T?` is equivalent to `[MaybeNull]T`; for argument values, `T?` is equivalent to `[AllowNull]T`. For more information, see the article on Attributes for null-state analysis in the language reference.

You can specify different behavior using constraints:

- The `class` constraint means that `T` must be a non-nullable reference type (for example `string`). The compiler produces a warning if you use a nullable reference type, such as `string?` for `T`.
- The `class?` constraint means that `T` must be a reference type, either non-nullable (`string`) or a nullable reference type (for example `string?`). When the type parameter is a nullable reference type, such as `string?`, an expression of `T?` references that same nullable reference type, such as `string?`.
- The `notnull` constraint means that `T` must be a non-nullable reference type, or a non-nullable value type. If you use a nullable reference type or a nullable value type for the type parameter, the compiler produces a warning. Furthermore, when `T` is a value type, the return value is that value type, not the corresponding nullable value type.

These constraints help provide more information to the compiler on how `T` is used. That helps when developers choose the type for `T` and provides better *null-state* analysis when an instance of the generic type is used.

Nullable context

The *nullable context* determines how nullable reference type annotations are handled and what warnings are produced by static null state analysis. The nullable context contains two flags: the *annotation* setting and the *warning* setting.

Both the *annotation* and *warning* settings are disabled by default for existing projects. Starting in .NET 6 (C# 10), both flags are enabled by default for *new* projects. The reason for two distinct flags for the nullable context is to make it easier to migrate large projects that predate the introduction of nullable reference types.

For small projects, you can enable nullable reference types, fix warnings, and continue. However, for larger projects and multi-project solutions, that might generate a large number of warnings. You can use pragmas to enable nullable reference types file-by-file as you begin using nullable reference types. The new features that protect against throwing a System.NullReferenceException can be disruptive when turned on in an existing codebase:

- All explicitly typed reference variables are interpreted as non-nullable reference types.
- The meaning of the `class` constraint in generics changed to mean a non-nullable reference type.
- New warnings are generated because of these new rules.

The **nullable annotation context** determines the compiler's behavior. There are four combinations for the **nullable context** settings:

- *both disabled*: The code is *nullable-oblivious*. *Disable* matches the behavior before nullable reference types were enabled, except the new syntax produces warnings instead of errors.
  - Nullable warnings are disabled.
  - All reference type variables are nullable reference types.
  - Use of the `?` suffix to declare a nullable reference type produces a warning.
  - You can use the null forgiving operator, `!`, but it has no effect.
- *both enabled*: The compiler enables all null reference analysis and all language features.
  - All new nullable warnings are enabled.
  - You can use the `?` suffix to declare a nullable reference type.
  - Reference type variables without the `?` suffix are non-nullable reference types.
  - The null forgiving operator suppresses warnings for a possible dereference of `null`.
- *warning enabled*: The compiler performs all null analysis and emits warnings when code might dereference `null`.
  - All new nullable warnings are enabled.
  - Use of the `?` suffix to declare a nullable reference type produces a warning.
  - All reference type variables are allowed to be null. However, members have the *null-state* of *not-null* at the opening brace of all methods unless declared with the `?` suffix.
  - You can use the null forgiving operator, `!`.
- *annotations enabled*: The compiler doesn't emit warnings when code might dereference `null`, or when you assign a maybe-null expression to a non-nullable variable.
  - All new nullable warnings are disabled.
  - You can use the `?` suffix to declare a nullable reference type.
  - Reference type variables without the `?` suffix are non-nullable reference types.
  - You can use the null forgiving operator, `!`, but it has no effect.

The nullable annotation context and nullable warning context can be set for a project using the <Nullable> element in your *.csproj* file. This element configures how the compiler interprets the nullability of types and what warnings are emitted. The following table shows the allowable values and summarizes the contexts they specify.

Expand table

Context	Dereference warnings	Assignment warnings	Reference types	`?` suffix	`!` operator
`disable`	Disabled	Disabled	All are nullable	Produces a warning	Has no effect
`enable`	Enabled	Enabled	Non-nullable unless declared with `?`	Declares nullable type	Suppresses warnings for possible `null` assignment
`warnings`	Enabled	Not applicable	All are nullable, but members are considered *not-null* at opening brace of methods	Produces a warning	Suppresses warnings for possible `null` assignment
`annotations`	Disabled	Disabled	Non-nullable unless declared with `?`	Declares nullable type	Has no effect

Reference type variables in code compiled in a *disabled* context are *nullable-oblivious*. You can assign a `null` literal or a *maybe-null* variable to a variable that is *nullable-oblivious*. However, the default state of a *nullable-oblivious* variable is *not-null*.

You can choose which setting is best for your project:

- Choose *disable* for legacy projects that you don't want to update based on diagnostics or new features.
- Choose *warnings* to determine where your code might throw System.NullReferenceExceptions. You can address those warnings before modifying code to enable non-nullable reference types.
- Choose *annotations* to express your design intent before enabling warnings.
- Choose *enable* for new projects and active projects where you want to protect against null reference exceptions.

Example:

XML

```
<Nullable>enable</Nullable>
```

You can also use directives to set these same flags anywhere in your source code. These directives are most useful when you're migrating a large codebase.

- `#nullable enable`: Sets the annotation and warning flags to **enable**.

- `#nullable disable`: Sets the annotation and warning flags to **disable**.
- `#nullable restore`: Restores the annotation flag and warning flag to the project settings.
- `#nullable disable warnings`: Set the warning flag to **disable**.
- `#nullable enable warnings`: Set the warning flag to **enable**.
- `#nullable restore warnings`: Restores the warning flag to the project settings.
- `#nullable disable annotations`: Set the annotation flag to **disable**.
- `#nullable enable annotations`: Set the annotation flag to **enable**.
- `#nullable restore annotations`: Restores the annotation flag to the project settings.

For any line of code, you can set any of the following combinations:

Expand table

Warning flag	Annotation flag	Use
project default	project default	Default
enable	disable	Fix analysis warnings
enable	project default	Fix analysis warnings
project default	enable	Add type annotations
enable	enable	Code already migrated
disable	enable	Annotate code before fixing warnings
disable	disable	Adding legacy code to migrated project
project default	disable	Rarely
disable	project default	Rarely

Those nine combinations provide you with fine-grained control over the diagnostics the compiler emits for your code. You can enable more features in any area you're updating, without seeing more warnings you aren't ready to address yet.

> **ⓘ Important**
>
> The global nullable context does not apply for generated code files. Under either strategy, the nullable context is *disabled* for any source file marked as generated. This means any APIs in generated files are not annotated. No nullable warnings are produced for generated files. There are four ways a file is marked as generated:

1. In the .editorconfig, specify `generated_code = true` in a section that applies to that file.
2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
3. Start the file name with *TemporaryGeneratedFile_*
4. End the file name with *.designer.cs*, *.generated.cs*, *.g.cs*, or *.g.i.cs*.

Generators can opt-in using the **#nullable** preprocessor directive.

By default, nullable annotation and warning flags are **disabled**. That means that your existing code compiles without changes and without generating any new warnings. Beginning with .NET 6, new projects include the `<Nullable>enable</Nullable>` element in all project templates, setting these flags to **enabled**.

These options provide two distinct strategies to update an existing codebase to use nullable reference types.

Known pitfalls

Arrays and structs that contain reference types are known pitfalls in nullable references and the static analysis that determines null safety. In both situations, a non-nullable reference might be initialized to `null`, without generating warnings.

Structs

A struct that contains non-nullable reference types allows assigning `default` for it without any warnings. Consider the following example:

C#

```
using System;

#nullable enable

public struct Student
{
    public string FirstName;
    public string? MiddleName;
    public string LastName;
}

public static class Program
```

```
{
    public static void PrintStudent(Student student)
    {
        Console.WriteLine($"First name: {student.FirstName.ToUpper()}");
        Console.WriteLine($"Middle name: {student.MiddleName?.ToUpper()}");
        Console.WriteLine($"Last name: {student.LastName.ToUpper()}");
    }

    public static void Main() => PrintStudent(default);
}
```

In the preceding example, there's no warning in `PrintStudent(default)` while the non-nullable reference types `FirstName` and `LastName` are null.

Another more common case is when you deal with generic structs. Consider the following example:

C#

```
#nullable enable

public struct S<T>
{
    public T Prop { get; set; }
}

public static class Program
{
    public static void Main()
    {
        string s = default(S<string>).Prop;
    }
}
```

In the preceding example, the property `Prop` is `null` at run time. It's assigned to non-nullable string without any warnings.

Arrays

Arrays are also a known pitfall in nullable reference types. Consider the following example that doesn't produce any warnings:

C#

```
using System;

#nullable enable
```

```
public static class Program
{
    public static void Main()
    {
        string[] values = new string[10];
        string s = values[0];
        Console.WriteLine(s.ToUpper());
    }
}
```

In the preceding example, the declaration of the array shows it holds non-nullable strings, while its elements are all initialized to `null`. Then, the variable `s` is assigned a `null` value (the first element of the array). Finally, the variable `s` is dereferenced causing a runtime exception.

Constructors

Constructor of a class will still call the finalizer, even when there was an exception thrown by that constructor.
The following example demonstrates that behavior:

C#

```
public class A
{
    private string _name;
    private B _b;

    public A(string name)
    {
        ArgumentNullException.ThrowIfNullOrEmpty(name);
        _name = name;
        _b = new B();
    }

  ~A()
  {
      Dispose();
  }

  public void Dispose()
  {
      _b.Dispose();
      GC.SuppressFinalize(this);
  }
}

public class B: IDisposable
{
    public void Dispose() { }
```

```
}

public void Main()
{
    var a = new A(string.Empty);
}
```

In the preceding example, the System.NullReferenceException will be thrown when `_b.Dispose();` runs, if the `name` parameter was `null`. The call to `_b.Dispose();` won't ever throw when the constructor completes successfully. However, there's no warning issued by the compiler, because static analysis can't determine if a method (like a constructor) completes without a runtime exception being thrown.

See also

- Nullable reference types specification
- Unconstrained type parameter annotations
- Intro to nullable references tutorial
- **Nullable** (C# Compiler option)
- CS8602: Possible dereference of null warning

Update a codebase with nullable reference types to improve null diagnostic warnings

Article • 09/21/2022

Nullable reference types enable you to declare if variables of a reference type should or shouldn't be assigned a `null` value. The compiler's static analysis and warnings when your code might dereference `null` are the most important benefit of this feature. Once enabled, the compiler generates warnings that help you avoid throwing a System.NullReferenceException when your code runs.

If your codebase is relatively small, you can turn on the feature in your project, address warnings, and enjoy the benefits of the improved diagnostics. Larger codebases may require a more structured approach to address warnings over time, enabling the feature for some as you address warnings in different types or files. This article describes different strategies to update a codebase and the tradeoffs associated with these strategies. Before starting your migration, read the conceptual overview of nullable reference types. It covers the compiler's static analysis, *null-state* values of *maybe-null* and *not-null* and the nullable annotations. Once you're familiar with those concepts and terms, you're ready to migrate your code.

Plan your migration

Regardless of how you update your codebase, the goal is that nullable warnings and nullable annotations are enabled in your project. Once you reach that goal, you'll have the `<nullable>Enable</nullable>` setting in your project. You won't need any of the preprocessor directives to adjust settings elsewhere.

> ⓘ **Note**
>
> You can designate a `Nullable` setting for your project using a `<Nullable>` tag. Refer to Compiler options for more information.

The first choice is setting the default for the project. Your choices are:

1. ***Nullable disable as the default***: *disable* is the default if you don't add a `Nullable` element to your project file. Use this default when you're not actively adding new files to the codebase. The main activity is to update the library to use nullable

reference types. Using this default means you add a nullable preprocessor directive to each file as you update its code.

2. ***Nullable enable as the default***: Set this default when you're actively developing new features. You want all new code to benefit from nullable reference types and nullable static analysis. Using this default means you must add a `#nullable disable` to the top of each file. You'll remove these preprocessor directives as you address the warnings in each file.
3. ***Nullable warnings as the default***: Choose this default for a two-phase migration. In the first phase, address warnings. In the second phase, turn on annotations for declaring a variable's expected *null-state*. Using this default means you must add a `#nullable disable` to the top of each file.
4. ***Nullable annotations*** as the default. Annotate code before addressing warnings.

Enabling nullable as the default creates more up-front work to add the preprocessor directives to every file. The advantage is that every new code file added to the project will be nullable enabled. Any new work will be nullable aware; only existing code must be updated. Disabling nullable as the default works better if the library is stable, and the main focus of the development is to adopt nullable reference types. You turn on nullable reference types as you annotate APIs. When you've finished, you enable nullable reference types for the entire project. When you create a new file, you must add the preprocessor directives and make it nullable aware. If any developers on your team forget, that new code is now in the backlog of work to make all code nullable aware.

Which of these strategies you pick depends on how much active development is taking place in your project. The more mature and stable your project, the better the second strategy. The more features being developed, the better the first strategy.

> **ⓘ Important**
>
> The global nullable context does not apply for generated code files. Under either strategy, the nullable context is *disabled* for any source file marked as generated. This means any APIs in generated files are not annotated. There are four ways a file is marked as generated:
>
> 1. In the .editorconfig, specify `generated_code = true` in a section that applies to that file.
> 2. Put `<auto-generated>` or `<auto-generated/>` in a comment at the top of the file. It can be on any line in that comment, but the comment block must be the first element in the file.
> 3. Start the file name with *TemporaryGeneratedFile_*
> 4. End the file name with *.designer.cs*, *.generated.cs*, *.g.cs*, or *.g.i.cs*.

Generators can opt-in using the #nullable preprocessor directive.

Understand contexts and warnings

Enabling warnings and annotations control how the compiler views reference types and nullability. Every type has one of three nullabilities:

- *oblivious*: All reference types are nullable *oblivious* when the annotation context is disabled.
- *nonnullable*: An unannotated reference type, `C` is *nonnullable* when the annotation context is enabled.
- *nullable*: An annotated reference type, `C?`, is *nullable*, but a warning may be issued when the annotation context is disabled. Variables declared with `var` are *nullable* when the annotation context is enabled.

The compiler generates warnings based on that nullability:

- *nonnullable* types cause warnings if a potential `null` value is assigned to them.
- *nullable* types cause warnings if they dereferenced when *maybe-null*.
- *oblivious* types cause warnings if they're dereferenced when *maybe-null* and the warning context is enabled.

Each variable has a default nullable state that depends on its nullability:

- Nullable variables have a default *null-state* of *maybe-null*.
- Non-nullable variables have a default *null-state* of *not-null*.
- Nullable oblivious variables have a default *null-state* of *not-null*.

Before you enable nullable reference types, all declarations in your codebase are *nullable oblivious*. That's important because it means all reference types have a default *null-state* of *not-null*.

Address warnings

If your project uses Entity Framework Core, you should read their guidance on Working with nullable reference types.

When you start your migration, you should start by enabling warnings only. All declarations remain *nullable oblivious*, but you'll see warnings when you dereference a value after its *null-state* changes to *maybe-null*. As you address these warnings, you'll be checking against null in more locations, and your codebase becomes more resilient. To

learn specific techniques for different situations, see the article on Techniques to resolve nullable warnings.

You can address warnings and enable annotations in each file or class before continuing with other code. However, it's often more efficient to address the warnings generated while the context is *warnings* before enabling the type annotations. That way, all types are *oblivious* until you've addressed the first set of warnings.

Enable type annotations

After addressing the first set of warnings, you can enable the *annotation context*. This changes reference types from *oblivious* to *nonnullable*. All variables declared with `var` are *nullable*. This change often introduces new warnings. The first step in addressing the compiler warnings is to use `?` annotations on parameter and return types to indicate when arguments or return values may be `null`. As you do this task, your goal isn't just to fix warnings. The more important goal is to make the compiler understand your intent for potential null values.

Attributes extend type annotations

Several attributes have been added to express additional information about the null state of variables. The rules for your APIs are likely more complicated than *not-null* or *maybe-null* for all parameters and return values. Many of your APIs have more complex rules for when variables can or can't be `null`. In these cases, you'll use attributes to express those rules. The attributes that describe the semantics of your API are found in the article on Attributes that affect nullable analysis.

Next steps

Once you've addressed all warnings after enabling annotations, you can set the default context for your project to *enabled*. If you added any pragmas in your code for the nullable annotation or warning context, you can remove them. Over time, you may see new warnings. You may write code that introduces warnings. A library dependency may be updated for nullable reference types. Those updates will change the types in that library from *nullable oblivious* to either *nonnullable* or *nullable*.

You can also explore these concepts in our Learn module on Nullable safety in C#.

Methods in C#

Article • 04/17/2025

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method.

> **Note**
>
> This article discusses named methods. For information about anonymous functions, see Lambda expressions.

Method signatures

Methods are declared in a `class`, `record`, or `struct` by specifying:

- An optional access level, such as `public` or `private`. The default is `private`.
- Optional modifiers such as `abstract` or `sealed`.
- The return value, or `void` if the method has none.
- The method name.
- Any method parameters. Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters.

These parts together form the method signature.

> **Important**
>
> A return type of a method isn't part of the signature of the method for the purposes of method overloading. However, it's part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

The following example defines a class named `Motorcycle` that contains five methods:

C#

```
namespace MotorCycleExample
{
    abstract class Motorcycle
    {
        // Anyone can call this.
        public void StartEngine() {/* Method statements here */ }
```

```
        // Only derived classes can call this.
        protected void AddGas(int gallons) { /* Method statements here */ }

        // Derived classes can override the base class implementation.
        public virtual int Drive(int miles, int speed) { /* Method statements here
*/ return 1; }

        // Derived classes can override the base class implementation.
        public virtual int Drive(TimeSpan time, int speed) { /* Method statements
here */ return 0; }

        // Derived classes must implement this.
        public abstract double GetTopSpeed();
    }
```

The `Motorcycle` class includes an overloaded method, `Drive`. Two methods have the same name, but their parameter lists differentiate them.

Method invocation

Methods can be either *instance* or *static*. You must instantiate an object to invoke an instance method on that instance; an instance method operates on that instance and its data. You invoke a static method by referencing the name of the type to which the method belongs; static methods don't operate on instance data. Attempting to call a static method through an object instance generates a compiler error.

Calling a method is like accessing a field. After the object name (if you're calling an instance method) or the type name (if you're calling a `static` method), add a period, the name of the method, and parentheses. Arguments are listed within the parentheses and are separated by commas.

The method definition specifies the names and types of any parameters that are required. When a caller invokes the method, it provides concrete values, called arguments, for each parameter. The arguments must be compatible with the parameter type, but the argument name, if one is used in the calling code, doesn't have to be the same as the parameter named defined in the method. In the following example, the `Square` method includes a single parameter of type `int` named *i*. The first method call passes the `Square` method a variable of type `int` named *num*; the second, a numeric constant; and the third, an expression.

C#

```
public static class SquareExample
{
    public static void Main()
    {
```

```
        // Call with an int variable.
        int num = 4;
        int productA = Square(num);

        // Call with an integer literal.
        int productB = Square(12);

        // Call with an expression that evaluates to int.
        int productC = Square(productA * 3);
    }

    static int Square(int i)
    {
        // Store input argument in a local variable.
        int input = i;
        return input * input;
    }
}
```

The most common form of method invocation used positional arguments; it supplies arguments in the same order as method parameters. The methods of the `Motorcycle` class can therefore be called as in the following example. The call to the `Drive` method, for example, includes two arguments that correspond to the two parameters in the method's syntax. The first becomes the value of the `miles` parameter. The second becomes the value of the `speed` parameter.

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        _ = moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine($"My top speed is {speed}");
    }
}
```

You can also use *named arguments* instead of positional arguments when invoking a method. When using named arguments, you specify the parameter name followed by a colon (":") and the argument. Arguments to the method can appear in any order, as long as all required arguments are present. The following example uses named arguments to invoke the

`TestMotorcycle.Drive` method. In this example, the named arguments are passed in the opposite order from the method's parameter list.

C#

```
namespace NamedMotorCycle;

class TestMotorcycle : Motorcycle
{
    public override int Drive(int miles, int speed) =>
        (int)Math.Round((double)miles / speed, 0);

    public override double GetTopSpeed() => 108.4;

    static void Main()
    {
        var moto = new TestMotorcycle();
        moto.StartEngine();
        moto.AddGas(15);
        int travelTime = moto.Drive(speed: 60, miles: 170);
        Console.WriteLine($"Travel time: approx. {travelTime} hours");
    }
}
// The example displays the following output:
//      Travel time: approx. 3 hours
```

You can invoke a method using both positional arguments and named arguments. However, positional arguments can only follow named arguments when the named arguments are in the correct positions. The following example invokes the `TestMotorcycle.Drive` method from the previous example using one positional argument and one named argument.

C#

```
int travelTime = moto.Drive(170, speed: 55);
```

Inherited and overridden methods

In addition to the members that are explicitly defined in a type, a type inherits members defined in its base classes. Since all types in the managed type system inherit directly or indirectly from the Object class, all types inherit its members, such as Equals(Object), GetType(), and ToString(). The following example defines a `Person` class, instantiates two `Person` objects, and calls the `Person.Equals` method to determine whether the two objects are equal. The `Equals` method, however, isn't defined in the `Person` class; it's inherited from Object.

C#

```
public class Person
{
    public string FirstName = default!;
}

public static class ClassTypeExample
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
    }
}
// The example displays the following output:
//       p1 = p2: False
```

Types can override inherited members by using the `override` keyword and providing an implementation for the overridden method. The method signature must be the same as the overridden method. The following example is like the previous one, except that it overrides the Equals(Object) method. (It also overrides the GetHashCode() method, since the two methods are intended to provide consistent results.)

C#

```
namespace methods;

public class Person
{
    public string FirstName = default!;

    public override bool Equals(object? obj) =>
        obj is Person p2 &&
        FirstName.Equals(p2.FirstName);

    public override int GetHashCode() => FirstName.GetHashCode();
}

public static class Example
{
    public static void Main()
    {
        Person p1 = new() { FirstName = "John" };
        Person p2 = new() { FirstName = "John" };
        Console.WriteLine($"p1 = p2: {p1.Equals(p2)}");
    }
}
// The example displays the following output:
//       p1 = p2: True
```

Passing parameters

Types in C# are either *value types* or *reference types*. For a list of built-in value types, see Types. By default, both value types and reference types are passed by value to a method.

Passing parameters by value

When a value type is passed to a method by value, a copy of the object instead of the object itself is passed to the method. Therefore, changes to the object in the called method have no effect on the original object when control returns to the caller.

The following example passes a value type to a method by value, and the called method attempts to change the value type's value. It defines a variable of type `int`, which is a value type, initializes its value to 20, and passes it to a method named `ModifyValue` that changes the variable's value to 30. When the method returns, however, the variable's value remains unchanged.

C#

```
public static class ByValueExample
{
    public static void Main()
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    static void ModifyValue(int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 20
```

When an object of a reference type is passed to a method by value, a reference to the object is passed by value. That is, the method receives not the object itself, but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the object when control returns to the calling method.

However, replacing the object passed to the method has no effect on the original object when control returns to the caller.

The following example defines a class (which is a reference type) named `SampleRefType`. It instantiates a `SampleRefType` object, assigns 44 to its `value` field, and passes the object to the `ModifyObject` method. This example does essentially the same thing as the previous example—it passes an argument by value to a method. But because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `obj.value` field also changes the `value` field of the argument, `rt`, in the `Main` method to 33, as the output from the example shows.

C#

```
public class SampleRefType
{
    public int value;
}

public static class ByRefTypeExample
{
    public static void Main()
    {
        var rt = new SampleRefType { value = 44 };
        ModifyObject(rt);
        Console.WriteLine(rt.value);
    }

    static void ModifyObject(SampleRefType obj) => obj.value = 33;
}
```

Passing parameters by reference

You pass a parameter by reference when you want to change the value of an argument in a method and want to reflect that change when control returns to the calling method. To pass a parameter by reference, you use the ref or out keyword. You can also pass a value by reference to avoid copying but still prevent modifications using the in keyword.

The following example is identical to the previous one except the value is passed by reference to the `ModifyValue` method. When the value of the parameter is modified in the `ModifyValue` method, the change in value is reflected when control returns to the caller.

C#

```
public static class ByRefExample
{
    public static void Main()
```

```
    {
        var value = 20;
        Console.WriteLine("In Main, value = {0}", value);
        ModifyValue(ref value);
        Console.WriteLine("Back in Main, value = {0}", value);
    }

    private static void ModifyValue(ref int i)
    {
        i = 30;
        Console.WriteLine("In ModifyValue, parameter value = {0}", i);
        return;
    }
}
// The example displays the following output:
//      In Main, value = 20
//      In ModifyValue, parameter value = 30
//      Back in Main, value = 30
```

A common pattern that uses by ref parameters involves swapping the values of variables. You pass two variables to a method by reference, and the method swaps their contents. The following example swaps integer values.

C#

```
public static class RefSwapExample
{
    static void Main()
    {
        int i = 2, j = 3;
        Console.WriteLine($"i = {i}  j = {j}");

        Swap(ref i, ref j);

        Console.WriteLine($"i = {i}  j = {j}");
    }

    static void Swap(ref int x, ref int y) =>
        (y, x) = (x, y);
}
// The example displays the following output:
//      i = 2  j = 3
//      i = 3  j = 2
```

Passing a reference-type parameter allows you to change the value of the reference itself, rather than the value of its individual elements or fields.

Parameter collections

Sometimes, the requirement that you specify the exact number of arguments to your method is restrictive. By using the `params` keyword to indicate that a parameter is a parameter collection, you allow your method to be called with a variable number of arguments. The parameter tagged with the `params` keyword must be a collection type, and it must be the last parameter in the method's parameter list.

A caller can then invoke the method in either of four ways for the `params` parameter:

- By passing a collection of the appropriate type that contains the desired number of elements. The example uses a collection expression so the compiler creates an appropriate collection type.
- By passing a comma-separated list of individual arguments of the appropriate type to the method. The compiler creates the appropriate collection type.
- By passing `null`.
- By not providing an argument to the parameter collection.

The following example defines a method named `GetVowels` that returns all the vowels from a parameter collection. The `Main` method illustrates all four ways of invoking the method. Callers aren't required to supply any arguments for parameters that include the `params` modifier. In that case, the parameter is an empty collection.

C#

```
static class ParamsExample
{
    static void Main()
    {
        string fromArray = GetVowels(["apple", "banana", "pear"]);
        Console.WriteLine($"Vowels from collection expression: '{fromArray}'");

        string fromMultipleArguments = GetVowels("apple", "banana", "pear");
        Console.WriteLine($"Vowels from multiple arguments:
'{fromMultipleArguments}'");

        string fromNull = GetVowels(null);
        Console.WriteLine($"Vowels from null: '{fromNull}'");

        string fromNoValue = GetVowels();
        Console.WriteLine($"Vowels from no value: '{fromNoValue}'");
    }

    static string GetVowels(params IEnumerable<string>? input)
    {
        if (input == null || !input.Any())
        {
            return string.Empty;
        }
```

```
        char[] vowels = ['A', 'E', 'I', 'O', 'U'];
        return string.Concat(
            input.SelectMany(
                word => word.Where(letter =>
vowels.Contains(char.ToUpper(letter)))));
    }
}

// The example displays the following output:
//     Vowels from array: 'aeaaaea'
//     Vowels from multiple arguments: 'aeaaaea'
//     Vowels from null: ''
//     Vowels from no value: ''
```

Before C# 13, the `params` modifier can be used only with a single dimensional array.

Optional parameters and arguments

A method definition can specify that its parameters are required or that they're optional. By default, parameters are required. Optional parameters are specified by including the parameter's default value in the method definition. When the method is called, if no argument is supplied for an optional parameter, the default value is used instead.

You assign the parameter's default value with one of the following kinds of expressions:

- A constant, such as a literal string or number.
- An expression of the form `default(SomeType)`, where `SomeType` can be either a value type or a reference type. If it's a reference type, it's effectively the same as specifying `null`. You can use the `default` literal, as the compiler can infer the type from the parameter's declaration.
- An expression of the form `new ValType()`, where `ValType` is a value type. This expression invokes the value type's implicit parameterless constructor, which isn't an actual member of the type.

  > **Note**
  >
  > When an expression of the form `new ValType()` invokes the explicitly defined parameterless constructor of a value type, the compiler generates an error as the default parameter value must be a compile-time constant. Use the `default(ValType)` expression or the `default` literal to provide the default parameter value. For more information about parameterless constructors, see the Struct initialization and default values section of the Structure types article.

If a method includes both required and optional parameters, optional parameters are defined at the end of the parameter list, after all required parameters.

The following example defines a method, `ExampleMethod`, that has one required and two optional parameters.

C#

```
public class Options
{
    public void ExampleMethod(int required, int optionalInt = default,
                              string? description = default)
    {
        var msg = $"{description ?? "N/A"}: {required} + {optionalInt} = {required
+ optionalInt}";
        Console.WriteLine(msg);
    }
}
```

The caller must supply an argument for all optional parameters up to the last optional parameter for which an argument is supplied. In the `ExampleMethod` method, for example, if the caller supplies an argument for the `description` parameter, it must also supply one for the `optionalInt` parameter. `opt.ExampleMethod(2, 2, "Addition of 2 and 2");` is a valid method call; `opt.ExampleMethod(2, , "Addition of 2 and 0");` generates an "Argument missing" compiler error.

If a method is called using named arguments or a combination of positional and named arguments, the caller can omit any arguments that follow the last positional argument in the method call.

The following example calls the `ExampleMethod` method three times. The first two method calls use positional arguments. The first omits both optional arguments, while the second omits the last argument. The third method call supplies a positional argument for the required parameter but uses a named argument to supply a value to the `description` parameter while omitting the `optionalInt` argument.

C#

```
public static class OptionsExample
{
    public static void Main()
    {
        var opt = new Options();
        opt.ExampleMethod(10);
        opt.ExampleMethod(10, 2);
        opt.ExampleMethod(12, description: "Addition with zero:");
    }
```

```
}
// The example displays the following output:
//      N/A: 10 + 0 = 10
//      N/A: 10 + 2 = 12
//      Addition with zero:: 12 + 0 = 12
```

The use of optional parameters affects *overload resolution*, or the way the C# compiler determines which overload to invoke for a method call, as follows:

- A member is a candidate for execution if each of its parameters corresponds by name or by position to a single argument. Furthermore, that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that doesn't have optional parameters for which arguments were omitted in the call.

Return values

Methods can return a value to the caller. If the return type (the type listed before the method name) isn't `void`, the method can return the value by using the `return` keyword. A statement with the `return` keyword followed by a variable, constant, or expression that matches the return type returns that value to the method caller. Methods with a nonvoid return type are required to use the `return` keyword to return a value. The `return` keyword also stops the execution of the method.

If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method stops executing when it reaches the end of the code block.

For example, these two methods use the `return` keyword to return integers:

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2) =>
        number1 + number2;

    public int SquareANumber(int number) =>
        number * number;
}
```

The preceding examples are expression bodied members. Expression bodied members return the value returned by the expression.

You can also choose to define your methods with a statement body and a `return` statement:

```
C#

class SimpleMathExtension
{
    public int DivideTwoNumbers(int number1, int number2)
    {
        return number1 / number2;
    }
}
```

To use a value returned from a method, you can assign the return value to a variable:

```
C#

int result = obj.DivideTwoNumbers(6,2);
// The result is 3.
Console.WriteLine(result);
```

The calling method can also use the method call itself anywhere a value of the same type would be sufficient. For example, the following two code examples accomplish the same goal:

```
C#

int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

```
C#

result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Sometimes, you want your method to return more than a single value. You use *tuple types* and *tuple literals* to return multiple values. The tuple type defines the data types of the tuple's elements. Tuple literals provide the actual values of the returned tuple. In the following example, `(string, string, string, int)` defines the tuple type returned by the `GetPersonalInfo` method. The expression `(per.FirstName, per.MiddleName, per.LastName,`

`per.Age)` is the tuple literal; the method returns the first, middle, and family name, along with the age, of a `PersonInfo` object.

C#

```
public (string, string, string, int) GetPersonalInfo(string id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The caller can then consume the returned tuple using the following code:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.Item1} {person.Item3}: age = {person.Item4}");
```

Names can also be assigned to the tuple elements in the tuple type definition. The following example shows an alternate version of the `GetPersonalInfo` method that uses named elements:

C#

```
public (string FName, string MName, string LName, int Age) GetPersonalInfo(string
id)
{
    PersonInfo per = PersonInfo.RetrieveInfoById(id);
    return (per.FirstName, per.MiddleName, per.LastName, per.Age);
}
```

The previous call to the `GetPersonalInfo` method can then be modified as follows:

C#

```
var person = GetPersonalInfo("111111111");
Console.WriteLine($"{person.FName} {person.LName}: age = {person.Age}");
```

If a method takes an array as a parameter and modifies the value of individual elements, it isn't necessary for the method to return the array. C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the following example, changes to the contents of the `values` array that are made in the `DoubleValues` method are observable by any code that has a reference to the array.

C#

```
public static class ArrayValueExample
{
    static void Main()
    {
        int[] values = [2, 4, 6, 8];
        DoubleValues(values);
        foreach (var value in values)
        {
            Console.Write("{0}  ", value);
        }
    }

    public static void DoubleValues(int[] arr)
    {
        for (var ctr = 0; ctr <= arr.GetUpperBound(0); ctr++)
        {
            arr[ctr] *= 2;
        }
    }
}
// The example displays the following output:
//       4  8  12  16
```

Extension methods

Ordinarily, there are two ways to add a method to an existing type:

- Modify the source code for that type. Modifying the source creates a breaking change if you also add any private data fields to support the method.
- Define the new method in a derived class. A method can't be added in this way using inheritance for other types, such as structures and enumerations. Nor can it be used to "add" a method to a sealed class.

Extension members let you "add" members to an existing type without modifying the type itself or implementing the new method in an inherited type. The extension member also doesn't have to reside in the same assembly as the type it extends. You call an extension method as if it were a defined member of a type.

For more information, see Extension members.

Async Methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an `await` expression in the async method, control returns to the caller if the awaited task isn't completed, and progress in the method with the `await` keyword is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

> **Note**
>
> An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method typically has a return type of Task<TResult>, Task, IAsyncEnumerable<T>, or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An async method that returns `void` can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws. An async method can have any task-like return type.

In the following example, `DelayAsync` is an async method that has a return statement that returns an integer. Because it's an async method, its method declaration must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer, as the following `int result = await delayTask` statement demonstrates.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();

    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
```

```
}
// Example output:
//   Result: 5
```

An async method can't declare any in, ref, or out parameters, but it can call methods that have such parameters.

For more information about async methods, see Asynchronous programming with async and await and Async return types.

Expression-bodied members

It's common to have method definitions that return immediately with the result of an expression, or that have a single statement as the body of the method. There's a syntax shortcut for defining such methods using `=>`:

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

If the method returns `void` or is an async method, the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read-only, and you don't use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator uses the yield return statement to return each element one at a time. When a `yield return` statement is reached, the current location is remembered so that the caller can request the next element in the sequence.

The return type of an iterator can be IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumerator, or IEnumerator<T>.

For more information, see Iterators.

See also

- Access Modifiers
- Static Classes and Static Class Members
- Inheritance
- Abstract and Sealed Classes and Class Members
- params
- out
- ref
- in
- Passing Parameters

Iterators

Article • 11/10/2021

Almost every program you write will have some need to iterate over a collection. You'll write code that examines every item in a collection.

You'll also create iterator methods, which are methods that produce an *iterator* for the elements of that class. An *iterator* is an object that traverses a container, particularly lists. Iterators can be used for:

- Performing an action on each item in a collection.
- Enumerating a custom collection.
- Extending LINQ or other libraries.
- Creating a data pipeline where data flows efficiently through iterator methods.

The C# language provides features for both generating and consuming sequences. These sequences can be produced and consumed synchronously or asynchronously. This article provides an overview of those features.

Iterating with foreach

Enumerating a collection is simple: The `foreach` keyword enumerates a collection, executing the embedded statement once for each element in the collection:

C#

```
foreach (var item in collection)
{
    Console.WriteLine(item?.ToString());
}
```

That's all. To iterate over all the contents of a collection, the `foreach` statement is all you need. The `foreach` statement isn't magic, though. It relies on two generic interfaces defined in the .NET core library to generate the code necessary to iterate a collection: `IEnumerable<T>` and `IEnumerator<T>`. This mechanism is explained in more detail below.

Both of these interfaces also have non-generic counterparts: `IEnumerable` and `IEnumerator`. The generic versions are preferred for modern code.

When a sequence is generated asynchronously, you can use the `await foreach` statement to asynchronously consume the sequence:

C#

```
await foreach (var item in asyncSequence)
{
Console.WriteLine(item?.ToString());
}
```

When a sequence is an System.Collections.Generic.IEnumerable<T>, you use `foreach`. When a sequence is an System.Collections.Generic.IAsyncEnumerable<T>, you use `await foreach`. In the latter case, the sequence is generated asynchronously.

Enumeration sources with iterator methods

Another great feature of the C# language enables you to build methods that create a source for an enumeration. These methods are referred to as *iterator methods*. An iterator method defines how to generate the objects in a sequence when requested. You use the `yield return` contextual keywords to define an iterator method.

You could write this method to produce the sequence of integers from 0 through 9:

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    yield return 0;
    yield return 1;
    yield return 2;
    yield return 3;
    yield return 4;
    yield return 5;
    yield return 6;
    yield return 7;
    yield return 8;
    yield return 9;
}
```

The code above shows distinct `yield return` statements to highlight the fact that you can use multiple discrete `yield return` statements in an iterator method. You can (and often do) use other language constructs to simplify the code of an iterator method. The method definition below produces the exact same sequence of numbers:

C#

```
public IEnumerable<int> GetSingleDigitNumbersLoop()
{
    int index = 0;
```

```
    while (index < 10)
        yield return index++;
}
```

You don't have to decide one or the other. You can have as many `yield return` statements as necessary to meet the needs of your method:

C#

```
public IEnumerable<int> GetSetsOfNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    index = 100;
    while (index < 110)
        yield return index++;
}
```

All of these preceding examples would have an asynchronous counterpart. In each case, you'd replace the return type of `IEnumerable<T>` with an `IAsyncEnumerable<T>`. For example, the previous example would have the following asynchronous version:

C#

```
public async IAsyncEnumerable<int> GetSetsOfNumbersAsync()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    await Task.Delay(500);

    yield return 50;

    await Task.Delay(500);

    index = 100;
    while (index < 110)
        yield return index++;
}
```

That's the syntax for both synchronous and asynchronous iterators. Let's consider a real world example. Imagine you're on an IoT project and the device sensors generate a very

large stream of data. To get a feel for the data, you might write a method that samples every Nth data element. This small iterator method does the trick:

C#

```
public static IEnumerable<T> Sample<T>(this IEnumerable<T> sourceSequence,
int interval)
{
    int index = 0;
    foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

If reading from the IoT device produces an asynchronous sequence, you'd modify the method as the following method shows:

C#

```
public static async IAsyncEnumerable<T> Sample<T>(this IAsyncEnumerable<T>
sourceSequence, int interval)
{
    int index = 0;
    await foreach (T item in sourceSequence)
    {
        if (index++ % interval == 0)
            yield return item;
    }
}
```

There's one important restriction on iterator methods: you can't have both a `return` statement and a `yield return` statement in the same method. The following code won't compile:

C#

```
public IEnumerable<int> GetSingleDigitNumbers()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    // generates a compile time error:
    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
};
```

```
    return items;
}
```

This restriction normally isn't a problem. You have a choice of either using `yield return` throughout the method, or separating the original method into multiple methods, some using `return`, and some using `yield return`.

You can modify the last method slightly to use `yield return` everywhere:

C#

```
public IEnumerable<int> GetFirstDecile()
{
    int index = 0;
    while (index < 10)
        yield return index++;

    yield return 50;

    var items = new int[] {100, 101, 102, 103, 104, 105, 106, 107, 108, 109
};
    foreach (var item in items)
        yield return item;
}
```

Sometimes, the right answer is to split an iterator method into two different methods. One that uses `return`, and a second that uses `yield return`. Consider a situation where you might want to return an empty collection, or the first five odd numbers, based on a boolean argument. You could write that as these two methods:

C#

```
public IEnumerable<int> GetSingleDigitOddNumbers(bool getCollection)
{
    if (getCollection == false)
        return new int[0];
    else
        return IteratorMethod();
}

private IEnumerable<int> IteratorMethod()
{
    int index = 0;
    while (index < 10)
    {
        if (index % 2 == 1)
            yield return index;
        index++;
```

```
    }
}
```

Look at the methods above. The first uses the standard `return` statement to return either an empty collection, or the iterator created by the second method. The second method uses the `yield return` statement to create the requested sequence.

Deeper dive into `foreach`

The `foreach` statement expands into a standard idiom that uses the `IEnumerable<T>` and `IEnumerator<T>` interfaces to iterate across all elements of a collection. It also minimizes errors developers make by not properly managing resources.

The compiler translates the `foreach` loop shown in the first example into something similar to this construct:

C#

```
IEnumerator<int> enumerator = collection.GetEnumerator();
while (enumerator.MoveNext())
{
    var item = enumerator.Current;
    Console.WriteLine(item.ToString());
}
```

The exact code generated by the compiler is more complicated, and handles situations where the object returned by `GetEnumerator()` implements the `IDisposable` interface. The full expansion generates code more like this:

C#

```
{
    var enumerator = collection.GetEnumerator();
    try
    {
        while (enumerator.MoveNext())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of enumerator.
    }
}
```

The compiler translates the first asynchronous sample into something similar to this construct:

C#

```
{
    var enumerator = collection.GetAsyncEnumerator();
    try
    {
        while (await enumerator.MoveNextAsync())
        {
            var item = enumerator.Current;
            Console.WriteLine(item.ToString());
        }
    }
    finally
    {
        // dispose of async enumerator.
    }
}
```

The manner in which the enumerator is disposed of depends on the characteristics of the type of `enumerator`. In the general synchronous case, the `finally` clause expands to:

C#

```
finally
{
   (enumerator as IDisposable)?.Dispose();
}
```

The general asynchronous case expands to:

C#

```
finally
{
    if (enumerator is IAsyncDisposable asyncDisposable)
        await asyncDisposable.DisposeAsync();
}
```

However, if the type of `enumerator` is a sealed type and there's no implicit conversion from the type of `enumerator` to `IDisposable` or `IAsyncDisposable`, the `finally` clause expands to an empty block:

C#

```
finally
{
}
```

If there's an implicit conversion from the type of `enumerator` to `IDisposable`, and `enumerator` is a non-nullable value type, the `finally` clause expands to:

C#

```
finally
{
   ((IDisposable)enumerator).Dispose();
}
```

Thankfully, you don't need to remember all these details. The `foreach` statement handles all those nuances for you. The compiler will generate the correct code for any of these constructs.

Introduction to delegates and events in C#

Article • 03/31/2022

Delegates provide a *late binding* mechanism in .NET. Late Binding means that you create an algorithm where the caller also supplies at least one method that implements part of the algorithm.

For example, consider sorting a list of stars in an astronomy application. You may choose to sort those stars by their distance from the earth, or the magnitude of the star, or their perceived brightness.

In all those cases, the Sort() method does essentially the same thing: arranges the items in the list based on some comparison. The code that compares two stars is different for each of the sort orderings.

These kinds of solutions have been used in software for half a century. The C# language delegate concept provides first class language support, and type safety around the concept.

As you'll see later in this series, the C# code you write for algorithms like this is type safe. The compiler ensures that the types match for arguments and return types.

Function pointers support similar scenarios, where you need more control over the calling convention. The code associated with a delegate is invoked using a virtual method added to a delegate type. Using function pointers, you can specify different conventions.

Language Design Goals for Delegates

The language designers enumerated several goals for the feature that eventually became delegates.

The team wanted a common language construct that could be used for any late binding algorithms. Delegates enable developers to learn one concept, and use that same concept across many different software problems.

Second, the team wanted to support both single and multicast method calls. (Multicast delegates are delegates that chain together multiple method calls. You'll see examples later in this series.)

The team wanted delegates to support the same type safety that developers expect from all C# constructs.

Finally, the team recognized an event pattern is one specific pattern where delegates, or any late binding algorithm, is useful. The team wanted to ensure the code for delegates could provide the basis for the .NET event pattern.

The result of all that work was the delegate and event support in C# and .NET.

The remaining articles in this series will cover language features, library support, and common idioms used when you work with delegates and events. You'll learn about:

- The `delegate` keyword and what code it generates.
- The features in the `System.Delegate` class, and how those features are used.
- How to create type-safe delegates.
- How to create methods that can be invoked through delegates.
- How to work with delegates and events by using lambda expressions.
- How delegates become one of the building blocks for LINQ.
- How delegates are the basis for the .NET event pattern, and how they're different.

Let's get started.

Next

System.Delegate and the `delegate` keyword

07/16/2025

Previous

This article covers the classes in .NET that support delegates, and how those map to the `delegate` keyword.

What are delegates?

Think of a delegate as a way to store a reference to a method, similar to how you might store a reference to an object. Just as you can pass objects to methods, you can pass method references using delegates. This is useful when you want to write flexible code where different methods can be "plugged in" to provide different behaviors.

For example, imagine you have a calculator that can perform operations on two numbers. Instead of hard-coding addition, subtraction, multiplication, and division into separate methods, you could use delegates to represent any operation that takes two numbers and returns a result.

Define delegate types

Now let's see how to create delegate types using the `delegate` keyword. When you define a delegate type, you're essentially creating a template that describes what kind of methods can be stored in that delegate.

You define a delegate type using syntax that looks similar to a method signature, but with the `delegate` keyword at the beginning:

C#

```
// Define a simple delegate that can point to methods taking two integers and
returning an integer
public delegate int Calculator(int x, int y);
```

This `Calculator` delegate can hold references to any method that takes two `int` parameters and returns an `int`.

Let's look at a more practical example. When you want to sort a list, you need to tell the sorting algorithm how to compare items. Let's see how delegates help with the `List.Sort()` method. The first step is to create a delegate type for the comparison operation:

C#

```
// From the .NET Core library
public delegate int Comparison<in T>(T left, T right);
```

This `Comparison<T>` delegate can hold references to any method that:

- Takes two parameters of type `T`
- Returns an `int` (typically -1, 0, or 1 to indicate "less than", "equal to", or "greater than")

When you define a delegate type like this, the compiler automatically generates a class derived from `System.Delegate` that matches your signature. This class handles all the complexity of storing and invoking the method references for you.

The `Comparison` delegate type is a generic type, which means it can work with any type `T`. For more information about generics, see Generic classes and methods.

Notice that even though the syntax looks similar to declaring a variable, you're actually declaring a new *type*. You can define delegate types inside classes, directly inside namespaces, or even in the global namespace.

> **Note**
>
> Declaring delegate types (or other types) directly in the global namespace is not recommended.

The compiler also generates add and remove handlers for this new type so that clients of this class can add and remove methods from an instance's invocation list. The compiler enforces that the signature of the method being added or removed matches the signature used when declaring the delegate type.

Declare instances of delegates

After defining the delegate type, you can create instances (variables) of that type. Think of this as creating a "slot" where you can store a reference to a method.

Like all variables in C#, you cannot declare delegate instances directly in a namespace or in the global namespace.

C#

```
// Inside a class definition:
```

```
public Comparison<T> comparator;
```

The type of this variable is `Comparison<T>` (the delegate type you defined earlier), and the name of the variable is `comparator`. At this point, `comparator` doesn't point to any method yet—it's like an empty slot waiting to be filled.

You can also declare delegate variables as local variables or method parameters, just like any other variable type.

Invoke delegates

Once you have a delegate instance that points to a method, you can call (invoke) that method through the delegate. You invoke the methods that are in the invocation list of a delegate by calling that delegate as if it were a method.

Here's how the `Sort()` method uses the comparison delegate to determine the order of objects:

```
C#
```

```
int result = comparator(left, right);
```

In this line, the code *invokes* the method attached to the delegate. You treat the delegate variable as if it were a method name and call it using normal method call syntax.

However, this line of code makes an unsafe assumption: it assumes that a target method has been added to the delegate. If no methods have been attached, the line above would cause a `NullReferenceException` to be thrown. The patterns used to address this problem are more sophisticated than a simple null-check, and are covered later in this series.

Assign, add, and remove invocation targets

Now you know how to define delegate types, declare delegate instances, and invoke delegates. But how do you actually connect a method to a delegate? This is where delegate assignment comes in.

To use a delegate, you need to assign a method to it. The method you assign must have the same signature (same parameters and return type) as the delegate type defines.

Let's see a practical example. Suppose you want to sort a list of strings by their length. You need to create a comparison method that matches the `Comparison<string>` delegate signature:

```
```

C#

```
private static int CompareLength(string left, string right) =>
    left.Length.CompareTo(right.Length);
```

This method takes two strings and returns an integer indicating which string is "greater" (longer in this case). The method is declared as private, which is perfectly fine. You don't need the method to be part of your public interface to use it with a delegate.

Now you can pass this method to the `List.Sort()` method:

C#

```
phrases.Sort(CompareLength);
```

Notice that you use the method name without parentheses. This tells the compiler to convert the method reference into a delegate that can be invoked later. The `Sort()` method will call your `CompareLength` method whenever it needs to compare two strings.

You can also be more explicit by declaring a delegate variable and assigning the method to it:

C#

```
Comparison<string> comparer = CompareLength;
phrases.Sort(comparer);
```

Both approaches accomplish the same thing. The first approach is more concise, while the second makes the delegate assignment more explicit.

For simple methods, it's common to use lambda expressions instead of defining a separate method:

C#

```
Comparison<string> comparer = (left, right) =>
left.Length.CompareTo(right.Length);
phrases.Sort(comparer);
```

Lambda expressions provide a compact way to define simple methods inline. Using lambda expressions for delegate targets is covered in more detail in a later section.

The examples so far show delegates with a single target method. However, delegate objects can support invocation lists that have multiple target methods attached to a single delegate object. This capability is particularly useful for event handling scenarios.

Delegate and MulticastDelegate classes

Behind the scenes, the delegate features you've been using are built on two key classes in the .NET framework: Delegate and MulticastDelegate. You don't usually work with these classes directly, but they provide the foundation that makes delegates work.

The `System.Delegate` class and its direct subclass `System.MulticastDelegate` provide the framework support for creating delegates, registering methods as delegate targets, and invoking all methods that are registered with a delegate.

Here's an interesting design detail: `System.Delegate` and `System.MulticastDelegate` are not themselves delegate types that you can use. Instead, they serve as the base classes for all the specific delegate types you create. The C# language prevents you from directly inheriting from these classes—you must use the `delegate` keyword instead.

When you use the `delegate` keyword to declare a delegate type, the C# compiler automatically creates a class derived from `MulticastDelegate` with your specific signature.

Why this design?

This design has its roots in the first release of C# and .NET. The design team had several goals:

1. **Type Safety**: The team wanted to ensure that the language enforced type safety when using delegates. This means ensuring that delegates are invoked with the correct type and number of arguments, and that return types are correctly verified at compile time.

2. **Performance**: By having the compiler generate concrete delegate classes that represent specific method signatures, the runtime can optimize delegate invocations.

3. **Simplicity**: Delegates were included in the 1.0 .NET release, which was before generics were introduced. The design needed to work within the constraints of the time.

The solution was to have the compiler create the concrete delegate classes that match your method signatures, ensuring type safety while hiding the complexity from you.

Working with delegate methods

Even though you can't create derived classes directly, you'll occasionally use methods defined on the `Delegate` and `MulticastDelegate` classes. Here are the most important ones to know about:

Every delegate you work with is derived from `MulticastDelegate`. A "multicast" delegate means that more than one method target can be invoked when calling through a delegate. The

original design considered making a distinction between delegates that could only invoke one method versus delegates that could invoke multiple methods. In practice, this distinction proved less useful than originally thought, so all delegates in .NET support multiple target methods.

The most commonly used methods when working with delegates are:

- `Invoke()`: Calls all the methods attached to the delegate
- `BeginInvoke()` / `EndInvoke()`: Used for asynchronous invocation patterns (though `async`/`await` is now preferred)

In most cases, you won't call these methods directly. Instead, you'll use the method call syntax on the delegate variable, as shown in the examples above. However, as you'll see later in this series, there are patterns that work directly with these methods.

Summary

Now that you've seen how the C# language syntax maps to the underlying .NET classes, you're ready to explore how strongly typed delegates are used, created, and invoked in more complex scenarios.

Next

Strongly Typed Delegates

Article • 09/15/2021

Previous

In the previous article, you saw that you create specific delegate types using the `delegate` keyword.

The abstract Delegate class provides the infrastructure for loose coupling and invocation. Concrete Delegate types become much more useful by embracing and enforcing type safety for the methods that are added to the invocation list for a delegate object. When you use the `delegate` keyword and define a concrete delegate type, the compiler generates those methods.

In practice, this would lead to creating new delegate types whenever you need a different method signature. This work could get tedious after a time. Every new feature requires new delegate types.

Thankfully, this isn't necessary. The .NET Core framework contains several types that you can reuse whenever you need delegate types. These are generic definitions so you can declare customizations when you need new method declarations.

The first of these types is the Action type, and several variations:

C#

```
public delegate void Action();
public delegate void Action<in T>(T arg);
public delegate void Action<in T1, in T2>(T1 arg1, T2 arg2);
// Other variations removed for brevity.
```

The `in` modifier on the generic type argument is covered in the article on covariance.

There are variations of the `Action` delegate that contain up to 16 arguments such as Action<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16>. It's important that these definitions use different generic arguments for each of the delegate arguments: That gives you maximum flexibility. The method arguments need not be, but may be, the same type.

Use one of the `Action` types for any delegate type that has a void return type.

The framework also includes several generic delegate types that you can use for delegate types that return values:

C#

```
public delegate TResult Func<out TResult>();
public delegate TResult Func<in T1, out TResult>(T1 arg);
public delegate TResult Func<in T1, in T2, out TResult>(T1 arg1, T2 arg2);
// Other variations removed for brevity
```

The `out` modifier on the result generic type argument is covered in the article on covariance.

There are variations of the `Func` delegate with up to 16 input arguments such as Func<T1,T2,T3,T4,T5,T6,T7,T8,T9,T10,T11,T12,T13,T14,T15,T16,TResult>. The type of the result is always the last type parameter in all the `Func` declarations, by convention.

Use one of the `Func` types for any delegate type that returns a value.

There's also a specialized Predicate<T> type for a delegate that returns a test on a single value:

C#

```
public delegate bool Predicate<in T>(T obj);
```

You may notice that for any `Predicate` type, a structurally equivalent `Func` type exists For example:

C#

```
Func<string, bool> TestForString;
Predicate<string> AnotherTestForString;
```

You might think these two types are equivalent. They are not. These two variables cannot be used interchangeably. A variable of one type cannot be assigned the other type. The C# type system uses the names of the defined types, not the structure.

All these delegate type definitions in the .NET Core Library should mean that you do not need to define a new delegate type for any new feature you create that requires delegates. These generic definitions should provide all the delegate types you need under most situations. You can simply instantiate one of these types with the required type parameters. In the case of algorithms that can be made generic, these delegates can be used as generic types.

This should save time, and minimize the number of new types that you need to create in order to work with delegates.

In the next article, you'll see several common patterns for working with delegates in practice.

Next

Common patterns for delegates

Article • 09/15/2021

Previous

Delegates provide a mechanism that enables software designs involving minimal coupling between components.

One excellent example for this kind of design is LINQ. The LINQ Query Expression Pattern relies on delegates for all of its features. Consider this simple example:

```
C#

var smallNumbers = numbers.Where(n => n < 10);
```

This filters the sequence of numbers to only those less than the value 10. The `Where` method uses a delegate that determines which elements of a sequence pass the filter. When you create a LINQ query, you supply the implementation of the delegate for this specific purpose.

The prototype for the Where method is:

```
C#

public static IEnumerable<TSource> Where<TSource> (this IEnumerable<TSource>
source, Func<TSource, bool> predicate);
```

This example is repeated with all the methods that are part of LINQ. They all rely on delegates for the code that manages the specific query. This API design pattern is a powerful one to learn and understand.

This simple example illustrates how delegates require very little coupling between components. You don't need to create a class that derives from a particular base class. You don't need to implement a specific interface. The only requirement is to provide the implementation of one method that is fundamental to the task at hand.

Build Your Own Components with Delegates

Let's build on that example by creating a component using a design that relies on delegates.

Let's define a component that could be used for log messages in a large system. The library components could be used in many different environments, on multiple different platforms. There are a lot of common features in the component that manages the logs. It will need to accept messages from any component in the system. Those messages will have different priorities, which the core component can manage. The messages should have timestamps in their final archived form. For more advanced scenarios, you could filter messages by the source component.

There is one aspect of the feature that will change often: where messages are written. In some environments, they may be written to the error console. In others, a file. Other possibilities include database storage, OS event logs, or other document storage.

There are also combinations of output that might be used in different scenarios. You may want to write messages to the console and to a file.

A design based on delegates will provide a great deal of flexibility, and make it easy to support storage mechanisms that may be added in the future.

Under this design, the primary log component can be a non-virtual, even sealed class. You can plug in any set of delegates to write the messages to different storage media. The built-in support for multicast delegates makes it easy to support scenarios where messages must be written to multiple locations (a file, and a console).

A First Implementation

Let's start small: the initial implementation will accept new messages, and write them using any attached delegate. You can start with one delegate that writes messages to the console.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(string msg)
    {
        if (WriteMessage is not null)
            WriteMessage(msg);
    }
}
```

The static class above is the simplest thing that can work. We need to write the single implementation for the method that writes messages to the console:

C#

```
public static class LoggingMethods
{
    public static void LogToConsole(string message)
    {
        Console.Error.WriteLine(message);
    }
}
```

Finally, you need to hook up the delegate by attaching it to the WriteMessage delegate declared in the logger:

C#

```
Logger.WriteMessage += LoggingMethods.LogToConsole;
```

Practices

Our sample so far is fairly simple, but it still demonstrates some of the important guidelines for designs involving delegates.

Using the delegate types defined in the core framework makes it easier for users to work with the delegates. You don't need to define new types, and developers using your library do not need to learn new, specialized delegate types.

The interfaces used are as minimal and as flexible as possible: To create a new output logger, you must create one method. That method may be a static method, or an instance method. It may have any access.

Format Output

Let's make this first version a bit more robust, and then start creating other logging mechanisms.

Next, let's add a few arguments to the `LogMessage()` method so that your log class creates more structured messages:

C#

```
public enum Severity
{
    Verbose,
    Trace,
    Information,
```

```
    Warning,
    Error,
    Critical
}
```

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static void LogMessage(Severity s, string component, string msg)
    {
        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

Next, let's make use of that `Severity` argument to filter the messages that are sent to the log's output.

C#

```
public static class Logger
{
    public static Action<string>? WriteMessage;

    public static Severity LogLevel { get; set; } = Severity.Warning;

    public static void LogMessage(Severity s, string component, string msg)
    {
        if (s < LogLevel)
            return;

        var outputMsg = $"{DateTime.Now}\t{s}\t{component}\t{msg}";
        if (WriteMessage is not null)
            WriteMessage(outputMsg);
    }
}
```

Practices

You've added new features to the logging infrastructure. Because the logger component is very loosely coupled to any output mechanism, these new features can be added with no impact on any of the code implementing the logger delegate.

As you keep building this, you'll see more examples of how this loose coupling enables greater flexibility in updating parts of the site without any changes to other locations. In fact, in a larger application, the logger output classes might be in a different assembly, and not even need to be rebuilt.

Build a Second Output Engine

The Log component is coming along well. Let's add one more output engine that logs messages to a file. This will be a slightly more involved output engine. It will be a class that encapsulates the file operations, and ensures that the file is always closed after each write. That ensures that all the data is flushed to disk after each message is generated.

Here is that file-based logger:

C#

```
public class FileLogger
{
    private readonly string logPath;
    public FileLogger(string path)
    {
        logPath = path;
        Logger.WriteMessage += LogMessage;
    }

    public void DetachLog() => Logger.WriteMessage -= LogMessage;
    // make sure this can't throw.
    private void LogMessage(string msg)
    {
        try
        {
            using (var log = File.AppendText(logPath))
            {
                log.WriteLine(msg);
                log.Flush();
            }
        }
        catch (Exception)
        {
            // Hmm. We caught an exception while
            // logging. We can't really log the
            // problem (since it's the log that's failing).
            // So, while normally, catching an exception
            // and doing nothing isn't wise, it's really the
            // only reasonable option here.
        }
    }
}
```

Once you've created this class, you can instantiate it and it attaches its LogMessage method to the Logger component:

```
C#

var file = new FileLogger("log.txt");
```

These two are not mutually exclusive. You could attach both log methods and generate messages to the console and a file:

```
C#

var fileOutput = new FileLogger("log.txt");
Logger.WriteMessage += LoggingMethods.LogToConsole; // LoggingMethods is the
static class we utilized earlier
```

Later, even in the same application, you can remove one of the delegates without any other issues to the system:

```
C#

Logger.WriteMessage -= LoggingMethods.LogToConsole;
```

Practices

Now, you've added a second output handler for the logging subsystem. This one needs a bit more infrastructure to correctly support the file system. The delegate is an instance method. It's also a private method. There's no need for greater accessibility because the delegate infrastructure can connect the delegates.

Second, the delegate-based design enables multiple output methods without any extra code. You don't need to build any additional infrastructure to support multiple output methods. They simply become another method on the invocation list.

Pay special attention to the code in the file logging output method. It is coded to ensure that it does not throw any exceptions. While this isn't always strictly necessary, it's often a good practice. If either of the delegate methods throws an exception, the remaining delegates that are on the invocation won't be invoked.

As a last note, the file logger must manage its resources by opening and closing the file on each log message. You could choose to keep the file open and implement `IDisposable` to close the file when you are completed. Either method has its advantages and disadvantages. Both do create a bit more coupling between the classes.

None of the code in the `Logger` class would need to be updated in order to support either scenario.

Handle Null Delegates

Finally, let's update the LogMessage method so that it is robust for those cases when no output mechanism is selected. The current implementation will throw a `NullReferenceException` when the `WriteMessage` delegate does not have an invocation list attached. You may prefer a design that silently continues when no methods have been attached. This is easy using the null conditional operator, combined with the `Delegate.Invoke()` method:

```
C#
```

```
public static void LogMessage(string msg)
{
    WriteMessage?.Invoke(msg);
}
```

The null conditional operator (`?.`) short-circuits when the left operand (`WriteMessage` in this case) is null, which means no attempt is made to log a message.

You won't find the `Invoke()` method listed in the documentation for `System.Delegate` or `System.MulticastDelegate`. The compiler generates a type safe `Invoke` method for any delegate type declared. In this example, that means `Invoke` takes a single `string` argument, and has a void return type.

Summary of Practices

You've seen the beginnings of a log component that could be expanded with other writers, and other features. By using delegates in the design, these different components are loosely coupled. This provides several advantages. It's easy to create new output mechanisms and attach them to the system. These other mechanisms only need one method: the method that writes the log message. It's a design that's resilient when new features are added. The contract required for any writer is to implement one method. That method could be a static or instance method. It could be public, private, or any other legal access.

The Logger class can make any number of enhancements or changes without introducing breaking changes. Like any class, you cannot modify the public API without the risk of breaking changes. But, because the coupling between the logger and any

output engines is only through the delegate, no other types (like interfaces or base classes) are involved. The coupling is as small as possible.

Next

Introduction to events

Article • 03/14/2025

Previous

Events are, like delegates, a *late binding* mechanism. In fact, events are built on the language support for delegates.

Events are a way for an object to broadcast (to all interested components in the system) that something happened. Any other component can subscribe to the event, and be notified when an event is raised.

You probably used events in some of your programming. Many graphical systems have an event model to report user interaction. These events would report mouse movement, button presses, and similar interactions. That's one of the most common, but not the only scenario where events are used.

You can define events that should be raised for your classes. One important consideration when working with events is that there might not be any object registered for a particular event. You must write your code so that it doesn't raise events when no listeners are configured.

Subscribing to an event also creates a coupling between two objects (the event source, and the event sink). You need to ensure that the event sink unsubscribes from the event source when no longer interested in events.

Design goals for event support

The language design for events targets these goals:

- Enable minimal coupling between an event source and an event sink. These two components might be written by different organizations, and might even be updated on different schedules.
- It should be simple to subscribe to an event, and to unsubscribe from that same event.
- Event sources should support multiple event subscribers. It should also support having no event subscribers attached.

You can see that the goals for events are similar to the goals for delegates. That's why the event language support is built on the delegate language support.

Language support for events

The syntax for defining events, and subscribing or unsubscribing from events is an extension of the syntax for delegates.

You use the `event` keyword to define an event:

```
C#

public event EventHandler<FileFoundArgs>? FileFound;
```

The type of the event (`EventHandler<FileListArgs>` in this example) must be a delegate type. There are conventions that you should follow when declaring an event. Typically, the event delegate type has a void return. Event declarations should be a verb, or a verb phrase. Use past tense when the event reports something that happened. Use a present tense verb (for example, `Closing`) to report something that is about to happen. Often, using present tense indicates that your class supports some kind of customization behavior. One of the most common scenarios is to support cancellation. For example, a `Closing` event can include an argument that would indicate if the close operation should continue, or not. Other scenarios enable callers to modify behavior by updating properties of the event arguments. You can raise an event to indicate a proposed next action an algorithm will take. The event handler might mandate a different action by modifying properties of the event argument.

When you want to raise the event, you call the event handlers using the delegate invocation syntax:

```
C#

FileFound?.Invoke(this, new FileFoundArgs(file));
```

As discussed in the section on delegates, the `?.` operator makes it easy to ensure that you don't attempt to raise the event when there are no subscribers to that event.

You subscribe to an event by using the `+=` operator:

```
C#

var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
```

```
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

The handler method typically has the prefix 'On' followed by the event name, as shown in the preceding code.

You unsubscribe using the `-=` operator:

```
C#

fileLister.FileFound -= onFileFound;
```

It's important that you declare a local variable for the expression that represents the event handler. That ensures the unsubscribe removes the handler. If, instead, you used the body of the lambda expression, you're attempting to remove a handler that was never attached, which does nothing.

In the next article, you'll learn more about typical event patterns, and different variations on this example.

Next

Standard .NET event patterns

Article • 03/15/2025

Previous

.NET events generally follow a few known patterns. Standardizing on these patterns means that developers can apply knowledge of those standard patterns, which can be applied to any .NET event program.

Let's go through these standard patterns so you have all the knowledge you need to create standard event sources, and subscribe and process standard events in your code.

Event delegate signatures

The standard signature for a .NET event delegate is:

```
C#

void EventRaised(object sender, EventArgs args);
```

This standard signature provides insight into when events are used:

- ***The return type is void***. Events can have zero to many listeners. Raising an event notifies all listeners. In general, listeners don't provide values in response to events.
- ***Events indicate the sender***: The event signature includes the object that raised the event. That provides any listener with a mechanism to communicate with the sender. The compile-time type of `sender` is `System.Object`, even though you likely know a more derived type that would always be correct. By convention, use `object`.
- ***Events package more information in a single structure***: The `args` parameter is a type derived from System.EventArgs that includes any more necessary information. (You'll see in the next section that this convention is no longer enforced.) If your event type doesn't need any more arguments, you still must provide both arguments. There's a special value, EventArgs.Empty that you should use to denote that your event doesn't contain any additional information.

Let's build a class that lists files in a directory, or any of its subdirectories that follow a pattern. This component raises an event for each file found that matches the pattern.

Using an event model provides some design advantages. You can create multiple event listeners that perform different actions when a sought file is found. Combining the

different listeners can create more robust algorithms.

Here's the initial event argument declaration for finding a sought file:

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

Even though this type looks like a small, data-only type, you should follow the convention and make it a reference (`class`) type. That means the argument object is passed by reference, and any updates to the data are viewed by all subscribers. The first version is an immutable object. You should prefer to make the properties in your event argument type immutable. That way, one subscriber can't change the values before another subscriber sees them. (There are exceptions to this practice, as you see later.)

Next, we need to create the event declaration in the FileSearcher class. Using the System.EventHandler<TEventArgs> type means that you don't need to create yet another type definition. You just use a generic specialization.

Let's fill out the FileSearcher class to search for files that match a pattern, and raise the correct event when a match is discovered.

C#

```
public class FileSearcher
{
    public event EventHandler<FileFoundArgs>? FileFound;

    public void Search(string directory, string searchPattern)
    {
        foreach (var file in Directory.EnumerateFiles(directory,
searchPattern))
        {
            FileFound?.Invoke(this, new FileFoundArgs(file));
        }
    }
}
```

Define and raise field-like events

The simplest way to add an event to your class is to declare that event as a public field, as in the preceding example:

C#

```
public event EventHandler<FileFoundArgs>? FileFound;
```

This looks like it's declaring a public field, which would appear to be a bad object-oriented practice. You want to protect data access through properties, or methods. While this code might look like a bad practice, the code generated by the compiler does create wrappers so that the event objects can only be accessed in safe ways. The only operations available on a field-like event are *add* and *remove* handler:

C#

```
var fileLister = new FileSearcher();
int filesFound = 0;

EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    filesFound++;
};

fileLister.FileFound += onFileFound;
```

C#

```
fileLister.FileFound -= onFileFound;
```

There's a local variable for the handler. If you used the body of the lambda, the `remove` handler wouldn't work correctly. It would be a different instance of the delegate, and silently do nothing.

Code outside the class can't raise the event, nor can it perform any other operations.

Beginning with C# 14, events can be declared as partial members. A partial event declaration must include a *defining declaration* and an *implementing declaration*. The defining declaration must use the field-like event syntax. The implementing declaration must declare the `add` and `remove` handlers.

Return values from event subscribers

Your simple version is working fine. Let's add another feature: Cancellation.

When you raise the *Found* event, listeners should be able to stop further processing, if this file is the last one sought.

The event handlers don't return a value, so you need to communicate that in another way. The standard event pattern uses the `EventArgs` object to include fields that event subscribers can use to communicate cancel.

Two different patterns could be used, based on the semantics of the Cancel contract. In both cases, you add a boolean field to the EventArguments for the found file event.

One pattern would allow any one subscriber to cancel the operation. For this pattern, the new field is initialized to `false`. Any subscriber can change it to `true`. After the raising the event for all subscribers, the FileSearcher component examines the boolean value and takes action.

The second pattern would only cancel the operation if all subscribers wanted the operation canceled. In this pattern, the new field is initialized to indicate the operation should cancel, and any subscriber could change it to indicate the operation should continue. After all subscribers process the raised the event, the FileSearcher component examines the boolean and takes action. There's one extra step in this pattern: the component needs to know if any subscribers responded to the event. If there are no subscribers, the field would indicate a cancel incorrectly.

Let's implement the first version for this sample. You need to add a boolean field named `CancelRequested` to the `FileFoundArgs` type:

C#

```
public class FileFoundArgs : EventArgs
{
    public string FoundFile { get; }
    public bool CancelRequested { get; set; }

    public FileFoundArgs(string fileName) => FoundFile = fileName;
}
```

This new field is automatically initialized to `false` so you don't cancel accidentally. The only other change to the component is to check the flag after raising the event to see if any of the subscribers requested a cancellation:

C#

```
private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
```

```
            var args = new FileFoundArgs(file);
            FileFound?.Invoke(this, args);
            if (args.CancelRequested)
                break;
        }
    }
```

One advantage of this pattern is that it isn't a breaking change. None of the subscribers requested cancellation before, and they still aren't. None of the subscriber code requires updates unless they want to support the new cancel protocol.

Let's update the subscriber so that it requests a cancellation once it finds the first executable:

C#

```
EventHandler<FileFoundArgs> onFileFound = (sender, eventArgs) =>
{
    Console.WriteLine(eventArgs.FoundFile);
    eventArgs.CancelRequested = true;
};
```

Adding another event declaration

Let's add one more feature, and demonstrate other language idioms for events. Let's add an overload of the `Search` method that traverses all subdirectories in search of files.

This method could get to be a lengthy operation in a directory with many subdirectories. Let's add an event that gets raised when each new directory search begins. This event enables subscribers to track progress, and update the user as to progress. All the samples you created so far are public. Let's make this event an internal event. That means you can also make the argument types internal as well.

You start by creating the new EventArgs derived class for reporting the new directory and progress.

C#

```
internal class SearchDirectoryArgs : EventArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs)
```

```
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

Again, you can follow the recommendations to make an immutable reference type for the event arguments.

Next, define the event. This time, you use a different syntax. In addition to using the field syntax, you can explicitly create the event property with add and remove handlers. In this sample, you don't need extra code in those handlers, but this shows how you would create them.

C#

```
internal event EventHandler<SearchDirectoryArgs> DirectoryChanged
{
    add { _directoryChanged += value; }
    remove { _directoryChanged -= value; }
}
private EventHandler<SearchDirectoryArgs>? _directoryChanged;
```

In many ways, the code you write here mirrors the code the compiler generates for the field event definitions you saw earlier. You create the event using syntax similar to properties. Notice that the handlers have different names: `add` and `remove`. These accessors are called to subscribe to the event, or unsubscribe from the event. Notice that you also must declare a private backing field to store the event variable. This variable is initialized to null.

Next, let's add the overload of the `Search` method that traverses subdirectories and raises both events. The easiest way is to use a default argument to specify that you want to search all directories:

C#

```
public void Search(string directory, string searchPattern, bool
searchSubDirs = false)
{
    if (searchSubDirs)
    {
        var allDirectories = Directory.GetDirectories(directory, "*.*",
SearchOption.AllDirectories);
        var completedDirs = 0;
        var totalDirs = allDirectories.Length + 1;
        foreach (var dir in allDirectories)
```

```
        {
            _directoryChanged?.Invoke(this, new (dir, totalDirs,
completedDirs++));
            // Search 'dir' and its subdirectories for files that match the
search pattern:
            SearchDirectory(dir, searchPattern);
        }
        // Include the Current Directory:
        _directoryChanged?.Invoke(this, new (directory, totalDirs,
completedDirs++));
        SearchDirectory(directory, searchPattern);
    }
    else
    {
        SearchDirectory(directory, searchPattern);
    }
}

private void SearchDirectory(string directory, string searchPattern)
{
    foreach (var file in Directory.EnumerateFiles(directory, searchPattern))
    {
        var args = new FileFoundArgs(file);
        FileFound?.Invoke(this, args);
        if (args.CancelRequested)
            break;
    }
}
```

At this point, you can run the application calling the overload for searching all subdirectories. There are no subscribers on the new `DirectoryChanged` event, but using the `?.Invoke()` idiom ensures it works correctly.

Let's add a handler to write a line that shows the progress in the console window.

C#

```
fileLister.DirectoryChanged += (sender, eventArgs) =>
{
    Console.Write($"Entering '{eventArgs.CurrentSearchDirectory}'.");
    Console.WriteLine($" {eventArgs.CompletedDirs} of {eventArgs.TotalDirs}
completed...");
};
```

You saw patterns that are followed throughout the .NET ecosystem. By learning these patterns and conventions, you're writing idiomatic C# and .NET quickly.

See also

- Introduction to events
- Event design
- Handle and raise events

Next, you see some changes in these patterns in the most recent release of .NET.

Next

The updated .NET Core event pattern

Article • 03/14/2025

Previous

The previous article discussed the most common event patterns. .NET Core has a more relaxed pattern. In this version, the `EventHandler<TEventArgs>` definition no longer has the constraint that `TEventArgs` must be a class derived from `System.EventArgs`.

This increases flexibility for you, and is backwards compatible. Let's start with the flexibility. The implementation for System.EventArgs uses a method defined in System.Object one method: MemberwiseClone(), which creates a shallow copy of the object. That method must use reflection in order to implement its functionality for any class derived from `EventArgs`. That functionality is easier to create in a specific derived class. That effectively means that deriving from System.EventArgs is a constraint that limits your designs, but doesn't provide any extra benefit. In fact, you can change the definitions of `FileFoundArgs` and `SearchDirectoryArgs` so that they don't derive from `EventArgs`. The program works exactly the same.

You could also change the `SearchDirectoryArgs` to a struct, if you make one more change:

C#

```
internal struct SearchDirectoryArgs
{
    internal string CurrentSearchDirectory { get; }
    internal int TotalDirs { get; }
    internal int CompletedDirs { get; }

    internal SearchDirectoryArgs(string dir, int totalDirs, int
completedDirs) : this()
    {
        CurrentSearchDirectory = dir;
        TotalDirs = totalDirs;
        CompletedDirs = completedDirs;
    }
}
```

The extra change is to call the parameterless constructor before entering the constructor that initializes all the fields. Without that addition, the rules of C# would report that the properties are being accessed before being assigned.

You shouldn't change the `FileFoundArgs` from a class (reference type) to a struct (value type). The protocol for handling cancel requires that you pass event arguments by reference. If you made the same change, the file search class could never observe any changes made by any of the event subscribers. A new copy of the structure would be used for each subscriber, and that copy would be a different copy than the one seen by the file search object.

Next, let's consider how this change can be backwards compatible. The removal of the constraint doesn't affect any existing code. Any existing event argument types do still derive from `System.EventArgs`. Backwards compatibility is one major reason why they continue to derive from `System.EventArgs`. Any existing event subscribers are subscribers to an event that followed the classic pattern.

Following similar logic, any event argument type created now wouldn't have any subscribers in any existing codebases. New event types that don't derive from `System.EventArgs` doesn't break those codebases.

Events with Async subscribers

You have one final pattern to learn: How to correctly write event subscribers that call async code. The challenge is described in the article on async and await. Async methods can have a void return type, but that is discouraged. When your event subscriber code calls an async method, you have no choice but to create an `async void` method. The event handler signature requires it.

You need to reconcile this opposing guidance. Somehow, you must create a safe `async void` method. The basics of the pattern you need to implement are shown in the following code:

C#

```
worker.StartWorking += async (sender, eventArgs) =>
{
    try
    {
        await DoWorkAsync();
    }
    catch (Exception e)
    {
        //Some form of logging.
        Console.WriteLine($"Async task failure: {e.ToString()}");
        // Consider gracefully, and quickly exiting.
    }
};
```

First, notice that the handler is marked as an async handler. Because it's being assigned to an event handler delegate type, it has a void return type. That means you must follow the pattern shown in the handler, and not allow any exceptions to be thrown out of the context of the async handler. Because it doesn't return a task, there's no task that can report the error by entering the faulted state. Because the method is async, the method can't throw the exception. (The calling method continues execution because it's `async`.) The actual runtime behavior is defined differently for different environments. It might terminate the thread or the process that owns the thread, or leave the process in an indeterminate state. All of these potential outcomes are highly undesirable.

You should wrap the `await` expression for the async Task in your own try block. If it does cause a faulted task, you can log the error. If it's an error from which your application can't recover, you can exit the program quickly and gracefully

This article explained the major updates to the .NET event pattern. You might see many examples of the earlier versions in the libraries you work with. However, you should understand what the latest patterns are as well. You can see the finished code for the sample at Program.cs.

The next article in this series helps you distinguish between using `delegates` and `events` in your designs. They're similar concepts, and that article helps you make the best decision for your programs.

Next

Distinguishing Delegates and Events

Article • 03/14/2025

Previous

Developers that are new to the .NET platform often struggle when deciding between a design based on `delegates` and a design based on `events`. The choice of delegates or events is often difficult, because the two language features are similar. Events are even built using the language support for delegates. An event handler declaration declares a delegate type.

Both offer a late binding scenario: they enable scenarios where a component communicates by calling a method that is only known at run time. They both support single and multiple subscriber methods. You might find these terms referred to as single cast and multicast support. They both support similar syntax for adding and removing handlers. Finally, raising an event and calling a delegate use exactly the same method call syntax. They even both support the same `Invoke()` method syntax for use with the `?.` operator.

With all those similarities, it's easy to have trouble determining when to use which.

Listening to Events is Optional

The most important consideration in determining which language feature to use is whether or not there must be an attached subscriber. If your code must call the code supplied by the subscriber, you should use a design based on delegates when you need to implement callback. If your code can complete all its work without calling any subscribers, you should use a design based on events.

Consider the examples built during this section. The code you built using `List.Sort()` must be given a comparer function in order to properly sort the elements. LINQ queries must be supplied with delegates in order to determine what elements to return. Both used a design built with delegates.

Consider the `Progress` event. It reports progress on a task. The task continues to proceed whether or not there are any listeners. The `FileSearcher` is another example. It would still search and find all the files that were sought, even with no event subscribers attached. UX controls still work correctly, even when there are no subscribers listening to the events. They both use designs based on events.

Return Values Require Delegates

Another consideration is the method prototype you would want for your delegate method. As you saw, the delegates used for events all have a void return type. There are idioms to create event handlers that do pass information back to event sources through modifying properties of the event argument object. While these idioms do work, they aren't as natural as returning a value from a method.

Notice that these two heuristics can often both be present: If your delegate method returns a value, it affects the algorithm in some way.

Events Have Private Invocation

Classes other than the one in which an event is contained can only add and remove event listeners; only the class containing the event can invoke the event. Events are typically public class members. By comparison, delegates are often passed as parameters and stored as private class members, if they're stored at all.

Event Listeners Often Have Longer Lifetimes

The longer lifetime of event listeners is a slightly weaker justification. However, you might find that event-based designs are more natural when the event source is raising events over a long period of time. You can see examples of event-based design for UX controls on many systems. Once you subscribe to an event, the event source can raise events throughout the lifetime of the program. (You can unsubscribe from events when you no longer need them.)

Contrast that with many delegate-based designs, where a delegate is used as an argument to a method, and the delegate isn't used after that method returns.

Evaluate Carefully

The above considerations aren't hard and fast rules. Instead, they represent guidance that can help you decide which choice is best for your particular usage. Because they're similar, you can even prototype both, and consider which would be more natural to work with. They both handle late binding scenarios well. Use the one that communicates your design the best.

Versioning in C#

07/09/2025

In this tutorial you'll learn what versioning means in .NET. You'll also learn the factors to consider when versioning your library as well as upgrading to a new version of a library.

Language version

The C# compiler is part of the .NET SDK. By default, the compiler chooses the C# language version that matches the chosen TFM for your project. If the SDK version is greater than your chosen framework, the compiler could use a greater language version. You can change the default by setting the `LangVersion` element in your project. You can learn how in our article on compiler options.

> ⚠ **Warning**
>
> Setting the `LangVersion` element to `latest` is discouraged. The `latest` setting means the installed compiler uses its latest version. That can change from machine to machine, making builds unreliable. In addition, it enables language features that may require runtime or library features not included in the current SDK.

Authoring Libraries

As a developer who has created .NET libraries for public use, you've most likely been in situations where you have to roll out new updates. How you go about this process matters a lot as you need to ensure that there's a seamless transition of existing code to the new version of your library. Here are several things to consider when creating a new release:

Semantic Versioning

Semantic versioning (SemVer for short) is a naming convention applied to versions of your library to signify specific milestone events. Ideally, the version information you give your library should help developers determine the compatibility with their projects that make use of older versions of that same library.

The most basic approach to SemVer is the 3 component format `MAJOR.MINOR.PATCH`, where:

- `MAJOR` is incremented when you make incompatible API changes
- `MINOR` is incremented when you add functionality in a backwards-compatible manner

- `PATCH` is incremented when you make backwards-compatible bug fixes

Understand version increments with examples

To help clarify when to increment each version number, here are concrete examples:

MAJOR version increments (incompatible API changes)

These changes require users to modify their code to work with the new version:

- Removing a public method or property:

```
C#

// Version 1.0.0
public class Calculator
{
    public int Add(int a, int b) => a + b;
    public int Subtract(int a, int b) => a - b; // This method exists
}

// Version 2.0.0 - MAJOR increment required
public class Calculator
{
    public int Add(int a, int b) => a + b;
    // Subtract method removed - breaking change!
}
```

- Changing method signatures:

```
C#

// Version 1.0.0
public void SaveFile(string filename) { }

// Version 2.0.0 - MAJOR increment required
public void SaveFile(string filename, bool overwrite) { } // Added required
parameter
```

- Changing the behavior of existing methods in ways that break expectations:

```
C#

// Version 1.0.0 - returns null when file not found
public string ReadFile(string path) => File.Exists(path) ?
File.ReadAllText(path) : null;

// Version 2.0.0 - MAJOR increment required
```

```
public string ReadFile(string path) => File.ReadAllText(path); // Now throws
exception when file not found
```

MINOR version increments (backwards-compatible functionality)

These changes add new features without breaking existing code:

- Adding new public methods or properties:

C#

```
// Version 1.0.0
public class Calculator
{
    public int Add(int a, int b) => a + b;
}

// Version 1.1.0 - MINOR increment
public class Calculator
{
    public int Add(int a, int b) => a + b;
    public int Multiply(int a, int b) => a * b; // New method added
}
```

- Adding new overloads:

C#

```
// Version 1.0.0
public void Log(string message) { }

// Version 1.1.0 - MINOR increment
public void Log(string message) { } // Original method unchanged
public void Log(string message, LogLevel level) { } // New overload added
```

- Adding optional parameters to existing methods:

C#

```
// Version 1.0.0
public void SaveFile(string filename) { }

// Version 1.1.0 - MINOR increment
public void SaveFile(string filename, bool overwrite = false) { } // Optional
parameter
```

> **Note**
>
> This is a *source compatible change*, but a *binary breaking change*. Users of this library must recompile for it to work correctly. Many libraries would consider this only in *major* version changes, not *minor* version changes.

PATCH version increments (backwards-compatible bug fixes)

These changes fix issues without adding new features or breaking existing functionality:

- Fixing a bug in an existing method's implementation:

```C#
// Version 1.0.0 - has a bug
public int Divide(int a, int b)
{
    return a / b; // Bug: doesn't handle division by zero
}

// Version 1.0.1 - PATCH increment
public int Divide(int a, int b)
{
    if (b == 0) throw new ArgumentException("Cannot divide by zero");
    return a / b; // Bug fixed, behavior improved but API unchanged
}
```

- Performance improvements that don't change the API:

```C#
// Version 1.0.0
public List<int> SortNumbers(List<int> numbers)
{
    return numbers.OrderBy(x => x).ToList(); // Slower implementation
}

// Version 1.0.1 - PATCH increment
public List<int> SortNumbers(List<int> numbers)
{
    var result = new List<int>(numbers);
    result.Sort(); // Faster implementation, same API
    return result;
}
```

The key principle is: if existing code can use your new version without any changes, it's a MINOR or PATCH update. If existing code needs to be modified to work with your new version,

it's a MAJOR update.

There are also ways to specify other scenarios, for example, pre-release versions, when applying version information to your .NET library.

Backwards Compatibility

As you release new versions of your library, backwards compatibility with previous versions will most likely be one of your major concerns. A new version of your library is source compatible with a previous version if code that depends on the previous version can, when recompiled, work with the new version. A new version of your library is binary compatible if an application that depended on the old version can, without recompilation, work with the new version.

Here are some things to consider when trying to maintain backwards compatibility with older versions of your library:

- Virtual methods: When you make a virtual method non-virtual in your new version it means that projects that override that method will have to be updated. This is a huge breaking change and is strongly discouraged.
- Method signatures: When updating a method behavior requires you to change its signature as well, you should instead create an overload so that code calling into that method will still work. You can always manipulate the old method signature to call into the new method signature so that implementation remains consistent.
- Obsolete attribute: You can use this attribute in your code to specify classes or class members that are deprecated and likely to be removed in future versions. This ensures developers utilizing your library are better prepared for breaking changes.
- Optional Method Arguments: When you make previously optional method arguments compulsory or change their default value then all code that does not supply those arguments will need to be updated.

> **Note**
>
> Making compulsory arguments optional should have very little effect especially if it doesn't change the method's behavior.

The easier you make it for your users to upgrade to the new version of your library, the more likely that they will upgrade sooner.

Application Configuration File

As a .NET developer there's a very high chance you've encountered the app.config file present in most project types. This simple configuration file can go a long way into improving the rollout of new updates. You should generally design your libraries in such a way that information that is likely to change regularly is stored in the `app.config` file, this way when such information is updated, the config file of older versions just needs to be replaced with the new one without the need for recompilation of the library.

Consuming Libraries

As a developer that consumes .NET libraries built by other developers you're most likely aware that a new version of a library might not be fully compatible with your project and you might often find yourself having to update your code to work with those changes.

Lucky for you, C# and the .NET ecosystem comes with features and techniques that allow us to easily update our app to work with new versions of libraries that might introduce breaking changes.

Assembly Binding Redirection

You can use the *app.config* file to update the version of a library your app uses. By adding what is called a *binding redirect*, you can use the new library version without having to recompile your app. The following example shows how you would update your app's *app.config* file to use the `1.0.1` patch version of `ReferencedLibrary` instead of the `1.0.0` version it was originally compiled with.

XML

```xml
<dependentAssembly>
    <assemblyIdentity name="ReferencedLibrary" publicKeyToken="32ab4ba45e0a69a1"
culture="en-us" />
    <bindingRedirect oldVersion="1.0.0" newVersion="1.0.1" />
</dependentAssembly>
```

> **Note**
>
> This approach will only work if the new version of `ReferencedLibrary` is binary compatible with your app. See the Backwards Compatibility section above for changes to look out for when determining compatibility.

new

You use the `new` modifier to hide inherited members of a base class. This is one way derived classes can respond to updates in base classes.

Take the following example:

C#

```
public class BaseClass
{
    public void MyMethod()
    {
        Console.WriteLine("A base method");
    }
}

public class DerivedClass : BaseClass
{
    public new void MyMethod()
    {
        Console.WriteLine("A derived method");
    }
}

public static void Main()
{
    BaseClass b = new BaseClass();
    DerivedClass d = new DerivedClass();

    b.MyMethod();
    d.MyMethod();
}
```

Output

Console

```
A base method
A derived method
```

From the example above you can see how `DerivedClass` hides the `MyMethod` method present in `BaseClass`. This means that when a base class in the new version of a library adds a member that already exists in your derived class, you can simply use the `new` modifier on your derived class member to hide the base class member.

When no `new` modifier is specified, a derived class will by default hide conflicting members in a base class, although a compiler warning will be generated the code will still compile. This means that simply adding new members to an existing class makes that new version of your library both source and binary compatible with code that depends on it.

override

The `override` modifier means a derived implementation extends the implementation of a base class member rather than hides it. The base class member needs to have the `virtual` modifier applied to it.

C#

```
public class MyBaseClass
{
    public virtual string MethodOne()
    {
        return "Method One";
    }
}

public class MyDerivedClass : MyBaseClass
{
    public override string MethodOne()
    {
        return "Derived Method One";
    }
}

public static void Main()
{
    MyBaseClass b = new MyBaseClass();
    MyDerivedClass d = new MyDerivedClass();

    Console.WriteLine($"Base Method One: {b.MethodOne()}");
    Console.WriteLine($"Derived Method One: {d.MethodOne()}");
}
```

Output

Console

```
Base Method One: Method One
Derived Method One: Derived Method One
```

The `override` modifier is evaluated at compile time and the compiler will throw an error if it doesn't find a virtual member to override.

Your knowledge of the discussed techniques and your understanding of the situations in which to use them, will go a long way towards easing the transition between versions of a library.

How to (C#)

In the How to section of the C# Guide, you can find quick answers to common questions. In some cases, articles may be listed in multiple sections. We wanted to make them easy to find for multiple search paths.

General C# concepts

There are several tips and tricks that are common C# developer practices:

- Initialize objects using an object initializer.
- Use operator overloading.
- Implement and call a custom extension member.
- Create a new method for an enum type using extension member.

Class, record, and struct members

You create classes, records, and structs to implement your program. These techniques are commonly used when writing classes, records, or structs.

- Declare automatically implemented properties.
- Declare and use read/write properties.
- Define constants.
- Override the ToString method to provide string output.
- Define abstract properties.
- Use the xml documentation features to document your code.
- Explicitly implement interface members to keep your public interface concise.
- Explicitly implement members of two interfaces.

Working with collections

These articles help you work with collections of data.

- Initialize a dictionary with a collection initializer.

Working with strings

Strings are the fundamental data type used to display or manipulate text. These articles demonstrate common practices with strings.

- Compare strings.

- Modify the contents of a string.
- Determine if a string represents a number.
- Use String.Split to separate strings.
- Combine multiple strings into one.
- Search for text in a string.

Convert between types

You may need to convert an object to a different type.

- Determine if a string represents a number.
- Convert between strings that represent hexadecimal numbers and the number.
- Convert a string to a DateTime.
- Convert a byte array to an int.
- Convert a string to a number.
- Use pattern matching, the as and is operators to safely cast to a different type.
- Define custom type conversions.
- Determine if a type is a nullable value type.
- Convert between nullable and non-nullable value types.

Equality and ordering comparisons

You may create types that define their own rules for equality or define a natural ordering among objects of that type.

- Test for reference-based equality.
- Define value-based equality for a type.

Exception handling

.NET programs report that methods did not successfully complete their work by throwing exceptions. In these articles you'll learn to work with exceptions.

- Handle exceptions using try and catch.
- Cleanup resources using finally clauses.
- Recover from non-CLS (Common Language Specification) exceptions.

Delegates and events

Delegates and events provide a capability for strategies that involve loosely coupled blocks of code.

- Declare, instantiate, and use delegates.
- Combine multicast delegates.

Events provide a mechanism to publish or subscribe to notifications.

- Subscribe and unsubscribe from events.
- Implement events declared in interfaces.
- Conform to .NET guidelines when your code publishes events.
- Raise events defined in base classes from derived classes.
- Implement custom event accessors.

LINQ practices

LINQ enables you to write code to query any data source that supports the LINQ query expression pattern. These articles help you understand the pattern and work with different data sources.

- Query a collection.
- Use var in query expressions.
- Return subsets of element properties from a query.
- Write queries with complex filtering.
- Sort elements of a data source.
- Sort elements on multiple keys.
- Control the type of a projection.
- Count occurrences of a value in a source sequence.
- Calculate intermediate values.
- Debug empty query results.
- Add custom methods to LINQ queries.

Multiple threads and async processing

Modern programs often use asynchronous operations. These articles will help you learn to use these techniques.

- Improve async performance using System.Threading.Tasks.Task.WhenAll.
- Make multiple web requests in parallel using async and await.
- Use a thread pool.

Command line args to your program

Typically, C# programs have command line arguments. These articles teach you to access and process those command line arguments.

- Retrieve all command line arguments with for.

Last updated on 11/26/2025

How to separate strings using String.Split in C#

The String.Split method creates an array of substrings by splitting the input string based on one or more delimiters. This method is often the easiest way to separate a string on word boundaries.

> **Tip**
>
> You can use AI assistance to **split a string**.

Split a string into words

The following code splits a common phrase into an array of strings for each word.

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
```

Every instance of a separator character produces a value in the returned array. Since arrays in C# are zero-indexed, each string in the array is indexed from 0 to the value returned by the Array.Length property minus 1:

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ');

for (int i = 0; i < words.Length; i++)
{
    Console.WriteLine($"Index {i}: <{words[i]}>");
}
```

The String.Split has many overloads. These overloads customize the behavior for splitting strings:

- You can specify separators as `char` values or `string` values.

- You can specify one separator or multiple separators. If you specify multiple separators, they must all be the same type (either `char` or `string`).
- You can specify the maximum number of substrings to return.
- You can specify if repeated separator characters are ignored, or produce empty substrings in the return value.
- You can specify if leading and trailing whitespace is removed from the returned substrings.

The remaining examples use different overloads to show each of these behaviors.

For more information about indices, see the Explore indexes and ranges article.

Specify multiple separators

String.Split can use multiple separator characters. The following example uses spaces, commas, periods, colons, and tabs as separating characters, which are passed to Split in an array. The loop at the bottom of the code displays each of the words in the returned array.

C#

```
char[] delimiterChars = [' ', ',', '.', ':', '\t'];

string text = "one\ttwo three:four,five six seven";
Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
```

Consecutive instances of any separator produce the empty string in the output array:

C#

```
char[] delimiterChars = [' ', ',', '.', ':', '\t'];

string text = "one\ttwo :,five six seven";
Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(delimiterChars);
Console.WriteLine($"{words.Length} words in text:");

foreach (var word in words)
{
```

```
    Console.WriteLine($"<{word}>");
}
```

String.Split can take an array of strings (character sequences that act as separators for parsing the target string, instead of single characters).

C#

```
string[] separatingStrings = ["<<", "..."];

string text = "one<<two......three<four";
Console.WriteLine($"Original text: '{text}'");

string[] words = text.Split(separatingStrings,
StringSplitOptions.RemoveEmptyEntries);
Console.WriteLine($"{words.Length} substrings in text:");

foreach (var word in words)
{
    Console.WriteLine(word);
}
```

Limit output size

The following example shows how to limit the output to the first four substrings in the source string.

C#

```
string phrase = "The quick brown fox jumps over the lazy dog.";
string[] words = phrase.Split(' ', 4, StringSplitOptions.None);

foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
```

Remove empty substrings

Consecutive separator characters produce the empty string as a value in the returned array. You can see how an empty string is created in the following example, which uses the space character as a separator.

C#

```
string phrase = "The quick brown    fox     jumps over the lazy dog.";
string[] words = phrase.Split(' ');
```

```
foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
```

This behavior makes it easier for formats like comma-separated values (CSV) files representing tabular data. Consecutive commas represent a blank column.

You can pass an optional StringSplitOptions.RemoveEmptyEntries parameter to exclude any empty strings in the returned array. For more complicated processing of the returned collection, you can use LINQ to manipulate the result sequence.

Trim whitespace

The following example shows the effect of trimming entries:

C#

```
string numerals = "1, 2, 3, 4, 5, 6, 7, 8, 9, 10";
string[] words = numerals.Split(',', StringSplitOptions.TrimEntries);

Console.WriteLine("Trimmed entries:");
foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
words = numerals.Split(',', StringSplitOptions.None);
Console.WriteLine("Untrimmed entries:");
foreach (var word in words)
{
    Console.WriteLine($"<{word}>");
}
```

The untrimmed entries have extra whitespace before the numerals.

Use AI to split a string

You can use AI tools, such as GitHub Copilot, to generate code to split strings using `String.Split` in C#. You can customize the prompt to use strings and delimiters per your requirements.

The following text shows an example prompt for Copilot Chat:

Copilot prompt

```
Generate C# code to parse and split CSV-style data from user input or external
export that isn't well formed.
The input string may contain mixed delimiters (commas, semicolons), inconsistent
spacing and empty entries.
Parse the string: accept all valid separators, remove any empty values and trim
whitespace from all entries.
Return a clean list of values.
Show example output for the string: "  apples, oranges ; bananas, ,  ;pears".
```

Review Copilot's suggestions before applying them.

For more information, see Copilot FAQs.

See also

- Extract elements from a string
- Strings
- .NET regular expressions
- GitHub Copilot in Visual Studio
- GitHub Copilot in VS Code

Last updated on 12/11/2025

How to concatenate multiple strings (C# Guide)

Concatenation is the process of appending one string to the end of another string. You concatenate strings by using the `+` operator. For string literals and string constants, concatenation occurs at compile time; no run-time concatenation occurs. For string variables, concatenation occurs only at run time.

> **Tip**
>
> You can use AI assistance to **concatenate strings**.

String literals

The following example splits a long string literal into smaller strings to improve readability in the source code. The code concatenates the smaller strings to create the long string literal. The parts are concatenated into a single string at compile time. There's no run-time performance cost regardless of the number of strings involved.

C#

```
// Concatenation of literals is performed at compile time, not run time.
string text = "Historically, the world of data and the world of objects " +
"have not been well integrated. Programmers work in C# or Visual Basic " +
"and also in SQL or XQuery. On the one side are concepts such as classes, " +
"objects, fields, inheritance, and .NET Framework APIs. On the other side " +
"are tables, columns, rows, nodes, and separate languages for dealing with " +
"them. Data types often require translation between the two worlds; there are " +
"different standard functions. Because the object world has no notion of query, a "
+
"query can only be represented as a string without compile-time type checking or " +
"IntelliSense support in the IDE. Transferring data from SQL tables or XML trees to
" +
"objects in memory is often tedious and error-prone.";

Console.WriteLine(text);
```

`+` and `+=` operators

To concatenate string variables, you can use the `+` or `+=` operators, string interpolation or the String.Format, String.Concat, String.Join or StringBuilder.Append methods. The `+` operator is easy to use and makes for intuitive code. Even if you use several `+` operators in one statement,

the string content is copied only once. The following code shows examples of using the `+` and `+=` operators to concatenate strings:

C#

```
string userName = "<Type your name here>";
string dateString = DateTime.Today.ToShortDateString();

// Use the + and += operators for one-time concatenations.
string str = "Hello " + userName + ". Today is " + dateString + ".";
Console.WriteLine(str);

str += " How are you today?";
Console.WriteLine(str);
```

String interpolation

In some expressions, it's easier to concatenate strings using string interpolation, as the following code shows:

C#

```
string userName = "<Type your name here>";
string date = DateTime.Today.ToShortDateString();

// Use string interpolation to concatenate strings.
string str = $"Hello {userName}. Today is {date}.";
Console.WriteLine(str);

str = $"{str} How are you today?";
Console.WriteLine(str);
```

> ⓘ Note
>
> In string concatenation operations, the C# compiler treats a null string the same as an empty string.

You can use string interpolation to initialize a constant string when all the expressions used for placeholders are also constant strings.

`String.Format`

Another method to concatenate strings is String.Format. This method works well when you're building a string from a few component strings.

`StringBuilder`

In other cases, you might be combining strings in a loop where the actual number of source strings can be large. The StringBuilder class was designed for these scenarios. The following code uses the Append method of the StringBuilder class to concatenate strings.

C#

```
// Use StringBuilder for concatenation in tight loops.
var sb = new StringBuilder();
for (int i = 0; i < 20; i++)
{
    sb.AppendLine(i.ToString());
}
Console.WriteLine(sb.ToString());
```

You can read more about the reasons to choose string concatenation or the StringBuilder class.

`String.Concat` or `String.Join`

Another option to join strings from a collection is to use String.Concat method. Use String.Join method if a delimiter should separate source strings. The following code combines an array of words using both methods:

C#

```
string[] words = ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy",
"dog."];

var unreadablePhrase = string.Concat(words);
Console.WriteLine(unreadablePhrase);

var readablePhrase = string.Join(" ", words);
Console.WriteLine(readablePhrase);
```

LINQ and `Enumerable.Aggregate`

At last, you can use LINQ and the Enumerable.Aggregate method to join strings from a collection. This method combines the source strings using a lambda expression. The lambda expression does the work to add each string to the existing accumulation. The following example combines an array of words, adding a space between each word in the array:

C#

```
string[] words = ["The", "quick", "brown", "fox", "jumps", "over", "the", "lazy",
"dog."];

var phrase = words.Aggregate((partialPhrase, word) =>$"{partialPhrase} {word}");
Console.WriteLine(phrase);
```

This option can cause more allocations than other methods for concatenating collections, as it creates an intermediate string for each iteration. If optimizing performance is critical, consider the StringBuilder class or the String.Concat or String.Join method to concatenate a collection, instead of `Enumerable.Aggregate`.

Use AI to concatenate strings

You can use AI tools, such as GitHub Copilot, to generate C# code to concatenate strings. You can customize the prompt to specify the output format, variables, and the method to use per your requirements.

Here's an example Copilot Chat prompt:

Copilot prompt

```
Generate C# code to use string interpolation to build an output string:
"Hello [name], today's date is [current date]. You have [count] new notifications."
Use variables for: name is "John", date is today's date, and notification count is
3.
Format the date as mm/dd/yyyy.
Show the output.
```

Review Copilot's suggestions before applying them.

For more information, see Copilot FAQs.

See also

- String
- StringBuilder
- Strings
- GitHub Copilot in Visual Studio
- GitHub Copilot in VS Code

Last updated on 12/11/2025

How to search strings

You can use two main strategies to search for text in strings. Methods of the String class search for specific text. Regular expressions search for patterns in text.

The string type, which is an alias for the System.String class, provides many useful methods for searching the contents of a string. Among them are Contains, StartsWith, EndsWith, IndexOf, LastIndexOf. The System.Text.RegularExpressions.Regex class provides a rich vocabulary to search for patterns in text. In this article, you learn these techniques and how to choose the best method for your needs.

Does a string contain text?

The String.Contains, String.StartsWith, and String.EndsWith methods search a string for specific text. The following example shows each of these methods and a variation that uses a case-insensitive search:

C#

```
string factMessage = "Extension methods have all the capabilities of regular static
methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// Simple comparisons are always case sensitive!
bool containsSearchResult = factMessage.Contains("extension");
// Raw string literals can work here because the output doesn't begin with "
Console.WriteLine($"""Contains "extension"? {containsSearchResult}""");

// For user input and strings that will be displayed to the end user,
// use the StringComparison parameter on methods that have it to specify how to
match strings.
bool ignoreCaseSearchResult = factMessage.StartsWith("extension",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"""Starts with "extension"? {ignoreCaseSearchResult} (ignoring
case)""");

bool endsWithSearchResult = factMessage.EndsWith(".",
System.StringComparison.CurrentCultureIgnoreCase);
Console.WriteLine($"Ends with '.'? {endsWithSearchResult}");
```

The preceding example demonstrates an important point for using these methods. Searches are **case-sensitive** by default. You use the StringComparison.CurrentCultureIgnoreCase enumeration value to specify a case-insensitive search.

Where does the sought text occur in a string?

The IndexOf and LastIndexOf methods also search for text in strings. These methods return the location of the text being sought. If the text isn't found, they return `-1`. The following example shows a search for the first and last occurrence of the word "methods" and displays the text in between.

C#

```
string factMessage = "Extension methods have all the capabilities of regular static methods.";

// Write the string and include the quotation marks.
Console.WriteLine($"\"{factMessage}\"");

// This search returns the substring between two strings, so
// the first index is moved to the character just after the first string.
int first = factMessage.IndexOf("methods") + "methods".Length;
int last = factMessage.LastIndexOf("methods");
string str2 = factMessage.Substring(first, last - first);
Console.WriteLine($"""Substring between "methods" and "methods": '{str2}'""");
```

Finding specific text using regular expressions

The System.Text.RegularExpressions.Regex class can be used to search strings. These searches can range in complexity from simple to complicated text patterns.

The following code example searches for the word "the" or "their" in a sentence, ignoring case. The static method Regex.IsMatch performs the search. You give it the string to search and a search pattern. In this case, a third argument specifies case-insensitive search. For more information, see System.Text.RegularExpressions.RegexOptions.

The search pattern describes the text you search for. The following table describes each element of the search pattern. (The following table uses the single `\`, which must be escaped as `\\` in a C# string).

Expand table

Pattern	Meaning
`the`	match the text "the"
`(eir)?`	match 0 or 1 occurrence of "eir"
`\s`	match a white-space character

C#

```
string[] sentences =
[
    "Put the water over there.",
    "They're quite thirsty.",
    "Their water bottles broke."
];

string sPattern = "the(ir)?\\s";

foreach (string s in sentences)
{
    Console.Write($"{s,24}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern,
System.Text.RegularExpressions.RegexOptions.IgnoreCase))
    {
        Console.WriteLine($"  (match for '{sPattern}' found)");
    }
    else
    {
        Console.WriteLine();
    }
}
```

> **Tip**
>
> The `string` methods are generally better choices when you're searching for an exact string. Regular expressions are better when you're searching for some pattern in a source string.

Does a string follow a pattern?

The following code uses regular expressions to validate the format of each string in an array. The validation requires that each string is formatted as a telephone number: three groups of digits separated by dashes where the first two groups contain three digits and the third group contains four digits. The search pattern uses the regular expression `^\\d{3}-\\d{3}-\\d{4}$`. For more information, see Regular Expression Language - Quick Reference.

Expand table

Pattern	Meaning
`^`	matches the beginning of the string

Pattern	Meaning
`\d{3}`	matches exactly three digit characters
`-`	matches the '-' character
`\d{4}`	matches exactly four digit characters
`$`	matches the end of the string

C#

```
string[] numbers =
[
    "123-555-0190",
    "444-234-22450",
    "690-555-0178",
    "146-893-232",
    "146-555-0122",
    "4007-555-0111",
    "407-555-0111",
    "407-2-5555",
    "407-555-8974",
    "407-2ab-5555",
    "690-555-8148",
    "146-893-232-"
];

string sPattern = """^\d{3}-\d{3}-\d{4}$""";

foreach (string s in numbers)
{
    Console.Write($"{s,14}");

    if (System.Text.RegularExpressions.Regex.IsMatch(s, sPattern))
    {
        Console.WriteLine(" - valid");
    }
    else
    {
        Console.WriteLine(" - invalid");
    }
}
```

This single search pattern matches many valid strings. Regular expressions are better to search for or validate against a pattern, rather than a single text string.

See also

- Strings

- System.Text.RegularExpressions.Regex
- .NET regular expressions
- Regular expression language - quick reference
- Best practices for using strings in .NET

Last updated on 12/11/2025

How to modify string contents in C#

This article demonstrates several techniques to produce a `string` by modifying an existing `string`. All the techniques demonstrated return the result of the modifications as a new `string` object. To demonstrate that the original and modified strings are distinct instances, the examples store the result in a new variable. You can examine the original `string` and the new, modified `string` when you run each example.

There are several techniques demonstrated in this article. You can replace existing text. You can search for patterns and replace matching text with other text. You can treat a string as a sequence of characters. You can also use convenience methods that remove white space. Choose the techniques that most closely match your scenario.

Replace text

The following code creates a new string by replacing existing text with a substitute.

C#

```
string source = "The mountains are behind the clouds today.";

// Replace one substring with another with String.Replace.
// Only exact matches are supported.
var replacement = source.Replace("mountains", "peaks");
Console.WriteLine($"The source string is <{source}>");
Console.WriteLine($"The updated string is <{replacement}>");
```

The preceding code demonstrates this *immutable* property of strings. You can see in the preceding example that the original string, `source`, isn't modified. The String.Replace method creates a new `string` containing the modifications.

The Replace method can replace either strings or single characters. In both cases, every occurrence of the sought text is replaced. The following example replaces all ' ' characters with '_':

C#

```
string source = "The mountains are behind the clouds today.";

// Replace all occurrences of one char with another.
var replacement = source.Replace(' ', '_');
Console.WriteLine(source);
Console.WriteLine(replacement);
```

The source string is unchanged, and a new string is returned with the replacement.

Trim white space

You can use the String.Trim, String.TrimStart, and String.TrimEnd methods to remove any leading or trailing white space. The following code shows an example of each. The source string doesn't change; these methods return a new string with the modified contents.

C#

```
// Remove trailing and leading white space.
string source = "    I'm wider than I need to be.      ";
// Store the results in a new string variable.
var trimmedResult = source.Trim();
var trimLeading = source.TrimStart();
var trimTrailing = source.TrimEnd();
Console.WriteLine($"<{source}>");
Console.WriteLine($"<{trimmedResult}>");
Console.WriteLine($"<{trimLeading}>");
Console.WriteLine($"<{trimTrailing}>");
```

Remove text

You can remove text from a string using the String.Remove method. This method removes the specified number of characters starting at a specific index. The following example shows how to use String.IndexOf followed by Remove to remove text from a string:

C#

```
string source = "Many mountains are behind many clouds today.";
// Remove a substring from the middle of the string.
string toRemove = "many ";
string result = string.Empty;
int i = source.IndexOf(toRemove);
if (i >= 0)
{
    result= source.Remove(i, toRemove.Length);
}
Console.WriteLine(source);
Console.WriteLine(result);
```

Replace matching patterns

You can use regular expressions to replace text matching patterns with new text, possibly defined by a pattern. The following example uses the System.Text.RegularExpressions.Regex class to find a pattern in a source string and replace it with proper capitalization. The Regex.Replace(String, String, MatchEvaluator, RegexOptions) method takes a function that

provides the logic of the replacement as one of its arguments. In this example, that function, `LocalReplaceMatchCase` is a **local function** declared inside the sample method. `LocalReplaceMatchCase` uses the System.Text.StringBuilder class to build the replacement string with proper capitalization.

Regular expressions are most useful for searching and replacing text that follows a pattern, rather than known text. For more information, see How to search strings. The search pattern, "the\s" searches for the word "the" followed by a white-space character. That part of the pattern ensures that it doesn't match "there" in the source string. For more information on regular expression language elements, see Regular Expression Language - Quick Reference.

C#

```
string source = "The mountains are still there behind the clouds today.";

// Use Regex.Replace for more flexibility.
// Replace "the" or "The" with "many" or "Many".
// using System.Text.RegularExpressions
string replaceWith = "many ";
source = System.Text.RegularExpressions.Regex.Replace(source, """the\s""",
LocalReplaceMatchCase,
    System.Text.RegularExpressions.RegexOptions.IgnoreCase);
Console.WriteLine(source);

string LocalReplaceMatchCase(System.Text.RegularExpressions.Match matchExpression)
{
    // Test whether the match is capitalized
    if (Char.IsUpper(matchExpression.Value[0]))
    {
        // Capitalize the replacement string
        System.Text.StringBuilder replacementBuilder = new
System.Text.StringBuilder(replaceWith);
        replacementBuilder[0] = Char.ToUpper(replacementBuilder[0]);
        return replacementBuilder.ToString();
    }
    else
    {
        return replaceWith;
    }
}
```

The StringBuilder.ToString method returns an immutable string with the contents in the StringBuilder object.

Modifying individual characters

You can produce a character array from a string, modify the contents of the array, and then create a new string from the modified contents of the array.

The following example shows how to replace a set of characters in a string. First, it uses the String.ToCharArray() method to create an array of characters. It uses the IndexOf method to find the starting index of the word "fox." The next three characters are replaced with a different word. Finally, a new string is constructed from the updated character array.

C#

```
string phrase = "The quick brown fox jumps over the fence";
Console.WriteLine(phrase);

char[] phraseAsChars = phrase.ToCharArray();
int animalIndex = phrase.IndexOf("fox");
if (animalIndex != -1)
{
    phraseAsChars[animalIndex++] = 'c';
    phraseAsChars[animalIndex++] = 'a';
    phraseAsChars[animalIndex] = 't';
}

string updatedPhrase = new string(phraseAsChars);
Console.WriteLine(updatedPhrase);
```

Programmatically build up string content

Since strings are immutable, the previous examples all create temporary strings or character arrays. In high-performance scenarios, it's desirable to avoid these heap allocations. .NET provides a String.Create method that allows you to programmatically fill in the character content of a string via a callback while avoiding the intermediate temporary string allocations.

C#

```
// constructing a string from a char array, prefix it with some additional
characters
char[] chars = [ 'a', 'b', 'c', 'd', '\0' ];
int length = chars.Length + 2;
string result = string.Create(length, chars, (Span<char> strContent, char[]
charArray) =>
{
        strContent[0] = '0';
        strContent[1] = '1';
        for (int i = 0; i < charArray.Length; i++)
        {
            strContent[i + 2] = charArray[i];
        }
});

Console.WriteLine(result);
```

You could modify a string in a fixed block with unsafe code, but it's **strongly** discouraged to modify the string content after a string is created. Doing so causes unpredictable bugs. For example, if someone interns a string that has the same content as yours, they get your copy and didn't expect that you're modifying their string.

See also

- .NET regular expressions
- Regular expression language - quick reference

Last updated on 12/11/2025

How to compare strings in C#

Article • 02/18/2025

You compare strings to answer one of two questions: "Are these two strings equal?" or "In what order should these strings be placed when sorting them?"

The following factors complicate these two questions:

- You can choose an ordinal or linguistic comparison.
- You can choose if case matters.
- You can choose culture-specific comparisons.
- Linguistic comparisons are culture and platform-dependent.

The System.StringComparison enumeration fields represent these choices:

- **CurrentCulture**: Compare strings using culture-sensitive sort rules and the current culture.
- **CurrentCultureIgnoreCase**: Compare strings using culture-sensitive sort rules, the current culture, and ignoring the case of the strings being compared.
- **InvariantCulture**: Compare strings using culture-sensitive sort rules and the invariant culture.
- **InvariantCultureIgnoreCase**: Compare strings using culture-sensitive sort rules, the invariant culture, and ignoring the case of the strings being compared.
- **Ordinal**: Compare strings using ordinal (binary) sort rules.
- **OrdinalIgnoreCase**: Compare strings using ordinal (binary) sort rules and ignoring the case of the strings being compared.

> **Note**
>
> The C# examples in this article run in the Try.NET inline code runner and playground. Select the **Run** button to run an example in an interactive window. Once you execute the code, you can modify it and run the modified code by selecting **Run** again. The modified code either runs in the interactive window or, if compilation fails, the interactive window displays all C# compiler error messages.

When you compare strings, you define an order among them. Comparisons are used to sort a sequence of strings. Once the sequence is in a known order, it's easier to search, both for software and for humans. Other comparisons might check if strings are the same. These sameness checks are similar to equality, but some differences, such as case differences, might be ignored.

Default ordinal comparisons

By default, the most common operations:

- String.Equals
- String.Equality and String.Inequality, that is, equality operators == and !=, respectively perform a case-sensitive, ordinal comparison. String.Equals has an overload where a StringComparison argument can be provided to alter its sorting rules. The following example demonstrates that:

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

result = root.Equals(root2, StringComparison.Ordinal);
Console.WriteLine($"Ordinal comparison: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");

Console.WriteLine($"Using == says that <{root}> and <{root2}> are {(root ==
root2 ? "equal" : "not equal")}");
```

The default ordinal comparison doesn't take linguistic rules into account when comparing strings. It compares the binary value of each Char object in two strings. As a result, the default ordinal comparison is also case-sensitive.

The test for equality with String.Equals and the `==` and `!=` operators differs from string comparison using the String.CompareTo and Compare(String, String) methods. They all perform a case-sensitive comparison. However, while the tests for equality perform an ordinal comparison, the `CompareTo` and `Compare` methods perform a culture-aware linguistic comparison using the current culture. Make the intent of your code clear by calling an overload that explicitly specifies the type of comparison to perform.

You can use the is operator and a constant pattern as an alternative to `==` when the right operand is a constant.

Case-insensitive ordinal comparisons

The String.Equals(String, StringComparison) method enables you to specify a StringComparison value of StringComparison.OrdinalIgnoreCase for a case-insensitive

ordinal comparison. There's also a static String.Compare(String, String, StringComparison) method that performs a case-insensitive ordinal comparison if you specify a value of StringComparison.OrdinalIgnoreCase for the StringComparison argument. These comparisons are shown in the following code:

C#

```
string root = @"C:\users";
string root2 = @"C:\Users";

bool result = root.Equals(root2, StringComparison.OrdinalIgnoreCase);
bool areEqual = String.Equals(root, root2,
StringComparison.OrdinalIgnoreCase);
int comparison = String.Compare(root, root2, comparisonType:
StringComparison.OrdinalIgnoreCase);

Console.WriteLine($"Ordinal ignore case: <{root}> and <{root2}> are {(result
? "equal." : "not equal.")}");
Console.WriteLine($"Ordinal static ignore case: <{root}> and <{root2}> are
{(areEqual ? "equal." : "not equal.")}");
if (comparison < 0)
{
    Console.WriteLine($"<{root}> is less than <{root2}>");
}
else if (comparison > 0)
{
    Console.WriteLine($"<{root}> is greater than <{root2}>");
}
else
{
    Console.WriteLine($"<{root}> and <{root2}> are equivalent in order");
}
```

These methods use the casing conventions of the invariant culture when performing a case-insensitive ordinal comparison.

Linguistic comparisons

Many string comparison methods (such as String.StartsWith) use linguistic rules for the *current culture* by default to order their inputs. This linguistic comparison is sometimes referred to as "word sort order." When you perform a linguistic comparison, some nonalphanumeric Unicode characters might have special weights assigned. For example, the hyphen "-" might have a small weight assigned to it so that "co-op" and "coop" appear next to each other in sort order. Some nonprinting control characters might be ignored. In addition, some Unicode characters might be equivalent to a sequence of Char instances. The following example uses the phrase "They dance in the street." in German with the "ss" (U+0073 U+0073) in one string and 'ß' (U+00DF) in another.

Linguistically (in Windows), "ss" is equal to the German Esszet: 'ß' character in both the "en-US" and "de-DE" cultures.

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

bool equal = String.Equals(first, second,
StringComparison.InvariantCulture);
Console.WriteLine($"The two strings {(equal == true ? "are" : "are not")}
equal.");
showComparison(first, second);

string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words);
showComparison(word, other);
showComparison(words, other);
void showComparison(string one, string two)
{
    int compareLinguistic = String.Compare(one, two,
StringComparison.InvariantCulture);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
    {
        Console.WriteLine($"<{one}> is less than <{two}> using invariant
culture");
    }
    else if (compareLinguistic > 0)
    {
        Console.WriteLine($"<{one}> is greater than <{two}> using invariant
culture");
    }
    else
    {
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using invariant culture");
    }

    if (compareOrdinal < 0)
    {
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    }
    else if (compareOrdinal > 0)
    {
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
```

```
    }
    else
    {
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
    }
}
```

On Windows, before .NET 5, the sort order of "cop", "coop", and "co-op" changes when you change from a linguistic comparison to an ordinal comparison. The two German sentences also compare differently using the different comparison types. Before .NET 5, the .NET globalization APIs used National Language Support (NLS) libraries. In .NET 5 and later versions, the .NET globalization APIs use International Components for Unicode (ICU) libraries, which unify .NET's globalization behavior across all supported operating systems.

Comparisons using specific cultures

The following example stores CultureInfo objects for the en-US and de-DE cultures. The comparisons are performed using a CultureInfo object to ensure a culture-specific comparison. The culture used affects linguistic comparisons. The following example shows the results of comparing the two German sentences using the "en-US" culture and the "de-DE" culture:

C#

```
string first = "Sie tanzen auf der Straße.";
string second = "Sie tanzen auf der Strasse.";

Console.WriteLine($"First sentence is <{first}>");
Console.WriteLine($"Second sentence is <{second}>");

var en = new System.Globalization.CultureInfo("en-US");

// For culture-sensitive comparisons, use the String.Compare
// overload that takes a StringComparison value.
int i = String.Compare(first, second, en,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {en.Name} returns {i}.");

var de = new System.Globalization.CultureInfo("de-DE");
i = String.Compare(first, second, de,
System.Globalization.CompareOptions.None);
Console.WriteLine($"Comparing in {de.Name} returns {i}.");

bool b = String.Equals(first, second, StringComparison.CurrentCulture);
Console.WriteLine($"The two strings {(b ? "are" : "are not")} equal.");
```

```
string word = "coop";
string words = "co-op";
string other = "cop";

showComparison(word, words, en);
showComparison(word, other, en);
showComparison(words, other, en);
void showComparison(string one, string two, System.Globalization.CultureInfo
culture)
{
    int compareLinguistic = String.Compare(one, two, en,
System.Globalization.CompareOptions.None);
    int compareOrdinal = String.Compare(one, two, StringComparison.Ordinal);
    if (compareLinguistic < 0)
    {
        Console.WriteLine($"<{one}> is less than <{two}> using en-US
culture");
    }
    else if (compareLinguistic > 0)
    {
        Console.WriteLine($"<{one}> is greater than <{two}> using en-US
culture");
    }
    else
    {
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using en-US culture");
    }

    if (compareOrdinal < 0)
    {
        Console.WriteLine($"<{one}> is less than <{two}> using ordinal
comparison");
    }
    else if (compareOrdinal > 0)
    {
        Console.WriteLine($"<{one}> is greater than <{two}> using ordinal
comparison");
    }
    else
    {
        Console.WriteLine($"<{one}> and <{two}> are equivalent in order
using ordinal comparison");
    }
}
```

Culture-sensitive comparisons are typically used to compare and sort strings input by users with other strings input by users. The characters and sorting conventions of these strings might vary depending on the locale of the user's computer. Even strings that contain identical characters might sort differently depending on the culture of the current thread.

Linguistic sorting and searching strings in arrays

The following examples show how to sort and search for strings in an array using a linguistic comparison dependent on the current culture. You use the static Array methods that take a System.StringComparer parameter.

The following example shows how to sort an array of strings using the current culture:

C#

```
string[] lines =
[
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
];

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

// Specify Ordinal to demonstrate the different behavior.
Array.Sort(lines, StringComparer.CurrentCulture);

foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Once the array is sorted, you can search for entries using a binary search. A binary search starts in the middle of the collection to determine which half of the collection would contain the sought string. Each subsequent comparison subdivides the remaining part of the collection in half. The array is sorted using the StringComparer.CurrentCulture. The local function `ShowWhere` displays information about where the string was found. If the string wasn't found, the returned value indicates where it would be if it were found.

C#

```
string[] lines =
[
    @"c:\public\textfile.txt",
```

```
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
];
Array.Sort(lines, StringComparer.CurrentCulture);

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = Array.BinarySearch(lines, searchString,
StringComparer.CurrentCulture);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");

void ShowWhere<T>(T[] array, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
        {
            Console.Write("beginning of sequence and ");
        }
        else
        {
            Console.Write($"{array[index - 1]} and ");
        }

        if (index == array.Length)
        {
            Console.WriteLine("end of sequence.");
        }
        else
        {
            Console.WriteLine($"{array[index]}.");
        }
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Ordinal sorting and searching in collections

The following code uses the System.Collections.Generic.List<T> collection class to store strings. The strings are sorted using the List<T>.Sort method. This method needs a

delegate that compares and orders two strings. The String.CompareTo method provides that comparison function. Run the sample and observe the order. This sort operation uses an ordinal case-sensitive sort. You would use the static String.Compare methods to specify different comparison rules.

C#

```
List<string> lines =
[
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
];

Console.WriteLine("Non-sorted order:");
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}

Console.WriteLine("\n\rSorted order:");

lines.Sort((left, right) => left.CompareTo(right));
foreach (string s in lines)
{
    Console.WriteLine($"   {s}");
}
```

Once sorted, the list of strings can be searched using a binary search. The following sample shows how to search the sorted list using the same comparison function. The local function `ShowWhere` shows where the sought text is or would be:

C#

```
List<string> lines =
[
    @"c:\public\textfile.txt",
    @"c:\public\textFile.TXT",
    @"c:\public\Text.txt",
    @"c:\public\testfile2.txt"
];
lines.Sort((left, right) => left.CompareTo(right));

string searchString = @"c:\public\TEXTFILE.TXT";
Console.WriteLine($"Binary search for <{searchString}>");
int result = lines.BinarySearch(searchString);
ShowWhere<string>(lines, result);

Console.WriteLine($"{(result > 0 ? "Found" : "Did not find")}
{searchString}");
```

```
void ShowWhere<T>(IList<T> collection, int index)
{
    if (index < 0)
    {
        index = ~index;

        Console.Write("Not found. Sorts between: ");

        if (index == 0)
        {
            Console.Write("beginning of sequence and ");
        }
        else
        {
            Console.Write($"{collection[index - 1]} and ");
        }

        if (index == collection.Count)
        {
            Console.WriteLine("end of sequence.");
        }
        else
        {
            Console.WriteLine($"{collection[index]}.");
        }
    }
    else
    {
        Console.WriteLine($"Found at index {index}.");
    }
}
```

Always make sure to use the same type of comparison for sorting and searching. Using different comparison types for sorting and searching produces unexpected results.

Collection classes such as System.Collections.Hashtable, System.Collections.Generic.Dictionary<TKey,TValue>, and System.Collections.Generic.List<T> have constructors that take a System.StringComparer parameter when the type of the elements or keys is `string`. In general, you should use these constructors whenever possible, and specify either StringComparer.Ordinal or StringComparer.OrdinalIgnoreCase.

See also

- System.Globalization.CultureInfo
- System.StringComparer
- Strings

- Comparing strings
- Globalizing and localizing applications

How to catch a non-CLS exception

Article • 09/15/2021

Some .NET languages, including C++/CLI, allow objects to throw exceptions that do not derive from Exception. Such exceptions are called *non-CLS exceptions* or *non-Exceptions*. In C# you cannot throw non-CLS exceptions, but you can catch them in two ways:

- Within a `catch (RuntimeWrappedException e)` block.

  By default, a Visual C# assembly catches non-CLS exceptions as wrapped exceptions. Use this method if you need access to the original exception, which can be accessed through the RuntimeWrappedException.WrappedException property. The procedure later in this topic explains how to catch exceptions in this manner.

- Within a general catch block (a catch block without an exception type specified) that is put after all other `catch` blocks.

  Use this method when you want to perform some action (such as writing to a log file) in response to non-CLS exceptions, and you do not need access to the exception information. By default the common language runtime wraps all exceptions. To disable this behavior, add this assembly-level attribute to your code, typically in the AssemblyInfo.cs file: `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`.

To catch a non-CLS exception

Within a `catch(RuntimeWrappedException e)` block, access the original exception through the RuntimeWrappedException.WrappedException property.

Example

The following example shows how to catch a non-CLS exception that was thrown from a class library written in C++/CLI. Note that in this example, the C# client code knows in advance that the exception type being thrown is a System.String. You can cast the RuntimeWrappedException.WrappedException property back its original type as long as that type is accessible from your code.

C#

```
// Class library written in C++/CLI.
var myClass = new ThrowNonCLS.Class1();

try
{
    // throws gcnew System::String(
    // "I do not derive from System.Exception!");
    myClass.TestThrow();
}
catch (RuntimeWrappedException e)
{
    String s = e.WrappedException as String;
    if (s != null)
    {
        Console.WriteLine(s);
    }
}
```

See also

- RuntimeWrappedException
- Exceptions and Exception Handling

Attributes

Article • 03/19/2025

Attributes provide a powerful way to associate metadata, or declarative information, with code (assemblies, types, methods, properties, and so on). After you associate an attribute with a program entity, you can query the attribute at run time by using a technique called *reflection*.

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 other required information.
- Attributes can be applied 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.
- Attributes enable a program to examine its own metadata or metadata in other programs by using reflection.

Work with reflection

Reflection APIs provided by Type 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. When you use attributes in your code, reflection enables you to access them. For more information, see Attributes.

Here's a simple example of reflection with the GetType() method. All types from the `Object` base class inherit this method, which is used to obtain the type of a variable:

> **Note**
>
> Make sure you add the `using System;` and `using System.Reflection;` statements at the top of your C# (*.cs*) code file.

C#

```
// Using GetType to obtain type information:
int i = 42;
```

```
Type type = i.GetType();
Console.WriteLine(type);
```

The output shows the type:

Output

```
System.Int32
```

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

C#

```
// Using Reflection to get information of an Assembly:
Assembly info = typeof(int).Assembly;
Console.WriteLine(info);
```

The output is similar to the following example:

Output

```
System.Private.CoreLib, Version=7.0.0.0, Culture=neutral,
PublicKeyToken=7cec85d7bea7798e
```

Keyword differences for IL

The C# keywords `protected` and `internal` have no meaning in Intermediate Language (IL) and aren't used in the reflection APIs. The corresponding terms in IL are *Family* and *Assembly*. Here some ways you can use these terms:

- To identify an `internal` method by using reflection, use the IsAssembly property.
- To identify a `protected internal` method, use the IsFamilyOrAssembly.

Work with attributes

Attributes can be placed on almost any declaration, though a specific attribute might restrict the types of declarations on which it's 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, you use the SerializableAttribute attribute to apply a specific characteristic to a class:

C#

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

You can declare a method with the DllImportAttribute attribute:

C#

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

You can place multiple attributes on a declaration:

C#

```
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. The following example shows multiuse of the ConditionalAttribute attribute:

C#

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

> **Note**
>
> By convention, all attribute names end with the suffix "Attribute" to distinguish them from other types in the .NET libraries. However, you don't need to specify the attribute suffix when you use attributes in code. For example, a `[DllImport]` declaration is equivalent to a `[DllImportAttribute]` declaration, but `DllImportAttribute` is the actual name of the class in the .NET Class Library.

Attribute parameters

Many attributes have parameters, which can be *positional*, *unnamed*, or *named*. The following table describes how to work with named and positional attributes:

Positional parameters

Parameters of the attribute constructor:

Named parameters

Properties or fields of the attribute:

- Must specify, can't omit
- Always specify first
- Specify in **certain** order
- Always optional, omit when false
- Specify after positional parameters
- Specify in any order

For example, the following code shows three equivalent `DllImport` attributes:

C#

```
[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 other instances are named parameters. In this scenario, both named parameters default to false, so they can be omitted. Refer to the individual attribute's documentation for information on default parameter values. For more information on allowed parameter types, see the Attributes section of the C# language specification.

Attribute targets

The *target* of an attribute is the entity that the attribute applies to. For example, an attribute can apply to a class, a method, or an assembly. By default, an attribute applies to the element that follows it. But you can also explicitly identify the element to associate, such as a method, a parameter, or the return value.

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

C#

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

The following table shows the list of possible `target` values.

Expand table

Target value	Applies to
`assembly`	Entire assembly
`module`	Current assembly module
`field`	Field in a class or a struct
`event`	Event
`method`	Method or `get` and `set` property accessors
`param`	Method parameters or `set` property accessor parameters
`property`	Property
`return`	Return value of a method, property indexer, or `get` property accessor
`type`	Struct, class, interface, enum, or delegate

You can specify the `field` target value to apply an attribute to the backing field created for an automatically implemented property.

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

C#

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

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

C#

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

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

```
// applies to parameter
int Method3([ValidatedContract] string contract) { return 0; }

// applies to return value
[return: ValidatedContract]
int Method4() { return 0; }
```

> **Note**
>
> Regardless of the targets on which the `ValidatedContract` attribute is defined to be valid, the `return` target has to be specified, even if the `ValidatedContract` attribute is defined to apply only to return values. In other words, the compiler doesn't use the `AttributeUsage` information to resolve ambiguous attribute targets. For more information, see **AttributeUsage**.

Review ways to use attributes

Here are some common ways to use attributes in code:

- Mark controller methods that respond to POST messages by using the `HttpPost` attribute. For more information, see the HttpPostAttribute class.
- Describe how to marshal method parameters when interoperating with native code. For more information, see the MarshalAsAttribute class.
- Describe Component Object Model (COM) properties for classes, methods, and interfaces.
- Call unmanaged code by using the DllImportAttribute class.
- Describe your assembly in terms of title, version, description, or trademark.
- Describe which members of a class to serialize for persistence.
- Describe how to map between class members and XML nodes for XML serialization.
- Describe the security requirements for methods.
- Specify characteristics used to enforce security.
- Control optimizations with the just-in-time (JIT) compiler so the code remains easy to debug.
- Obtain information about the caller to a method.

Review reflection scenarios

Reflection is useful in the following scenarios:

- Access attributes in your program's metadata. For more information, see Retrieving information stored in attributes.
- Examine and instantiate types in an assembly.
- Build new types at run time by using classes in the System.Reflection.Emit namespace.
- Perform late binding and access methods on types created at run time. For more information, see Dynamically loading and using types.

Related links

- Common attributes (C#)
- Caller information (C#)
- Attributes
- Reflection
- View type information
- Reflection and generic types
- System.Reflection.Emit
- Retrieving information stored in attributes

Create custom attributes

Article • 03/15/2023

You can create your own custom attributes by defining an attribute class, a class that derives directly or indirectly from Attribute, which makes identifying attribute definitions in metadata fast and easy. Suppose you want to tag types with the name of the programmer who wrote the type. You might define a custom `Author` attribute class:

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct)
]
public class AuthorAttribute : System.Attribute
{
    private string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;
        Version = 1.0;
    }
}
```

The class name `AuthorAttribute` is the attribute's name, `Author`, plus the `Attribute` suffix. It's derived from `System.Attribute`, so it's a custom attribute class. The constructor's parameters are the custom attribute's positional parameters. In this example, `name` is a positional parameter. Any public read-write fields or properties are named parameters. In this case, `version` is the only named parameter. Note the use of the `AttributeUsage` attribute to make the `Author` attribute valid only on class and `struct` declarations.

You could use this new attribute as follows:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass
{
    // P. Ackerman's code goes here...
}
```

`AttributeUsage` has a named parameter, `AllowMultiple`, with which you can make a custom attribute single-use or multiuse. In the following code example, a multiuse attribute is created.

C#

```
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}
```

In the following code example, multiple attributes of the same type are applied to a class.

C#

```
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}
```

See also

- System.Reflection
- Writing Custom Attributes
- AttributeUsage (C#)

Access attributes using reflection

Article • 03/15/2023

The fact that you can define custom attributes and place them in your source code would be of little value without some way of retrieving that information and acting on it. By using reflection, you can retrieve the information that was defined with custom attributes. The key method is `GetCustomAttributes`, which returns an array of objects that are the run-time equivalents of the source code attributes. This method has many overloaded versions. For more information, see Attribute.

An attribute specification such as:

C#

```
[Author("P. Ackerman", Version = 1.1)]
class SampleClass { }
```

is conceptually equivalent to the following code:

C#

```
var anonymousAuthorObject = new Author("P. Ackerman")
{
    Version = 1.1
};
```

However, the code isn't executed until `SampleClass` is queried for attributes. Calling `GetCustomAttributes` on `SampleClass` causes an `Author` object to be constructed and initialized. If the class has other attributes, other attribute objects are constructed similarly. `GetCustomAttributes` then returns the `Author` object and any other attribute objects in an array. You can then iterate over this array, determine what attributes were applied based on the type of each array element, and extract information from the attribute objects.

Here's a complete example. A custom attribute is defined, applied to several entities, and retrieved via reflection.

C#

```
// Multiuse attribute.
[System.AttributeUsage(System.AttributeTargets.Class |
                       System.AttributeTargets.Struct,
                       AllowMultiple = true)  // Multiuse attribute.
]
```

```
public class AuthorAttribute : System.Attribute
{
    string Name;
    public double Version;

    public AuthorAttribute(string name)
    {
        Name = name;

        // Default value.
        Version = 1.0;
    }

    public string GetName() => Name;
}

// Class with the Author attribute.
[Author("P. Ackerman")]
public class FirstClass
{
    // ...
}

// Class without the Author attribute.
public class SecondClass
{
    // ...
}

// Class with multiple Author attributes.
[Author("P. Ackerman"), Author("R. Koch", Version = 2.0)]
public class ThirdClass
{
    // ...
}

class TestAuthorAttribute
{
    public static void Test()
    {
        PrintAuthorInfo(typeof(FirstClass));
        PrintAuthorInfo(typeof(SecondClass));
        PrintAuthorInfo(typeof(ThirdClass));
    }

    private static void PrintAuthorInfo(System.Type t)
    {
        System.Console.WriteLine($"Author information for {t}");

        // Using reflection.
        System.Attribute[] attrs = System.Attribute.GetCustomAttributes(t);
// Reflection.

        // Displaying output.
        foreach (System.Attribute attr in attrs)
```

```
        {
            if (attr is AuthorAttribute a)
            {
                System.Console.WriteLine($"   {a.GetName()}, version
{a.Version:f}");
            }
        }
    }
}
/* Output:
    Author information for FirstClass
       P. Ackerman, version 1.00
    Author information for SecondClass
    Author information for ThirdClass
       R. Koch, version 2.00
       P. Ackerman, version 1.00
*/
```

See also

- System.Reflection
- Attribute
- Retrieving Information Stored in Attributes

How to create a C/C++ union by using attributes in C#

Article • 03/15/2023

By using attributes, you can customize how structs are laid out in memory. For example, you can create what is known as a union in C/C++ by using the `StructLayout(LayoutKind.Explicit)` and `FieldOffset` attributes.

In this code segment, all of the fields of `TestUnion` start at the same location in memory.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestUnion
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public double d;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public byte b;
}
```

The following code is another example where fields start at different explicitly set locations.

C#

```
[System.Runtime.InteropServices.StructLayout(LayoutKind.Explicit)]
struct TestExplicit
{
    [System.Runtime.InteropServices.FieldOffset(0)]
    public long lg;

    [System.Runtime.InteropServices.FieldOffset(0)]
    public int i1;

    [System.Runtime.InteropServices.FieldOffset(4)]
    public int i2;

    [System.Runtime.InteropServices.FieldOffset(8)]
    public double d;
```

```
    [System.Runtime.InteropServices.FieldOffset(12)]
    public char c;

    [System.Runtime.InteropServices.FieldOffset(14)]
    public byte b;
}
```

The two integer fields, `i1` and `i2` combined, share the same memory locations as `lg`. Either `lg` uses the first 8 bytes, or `i1` uses the first 4 bytes and `i2` uses the next 4 bytes. This sort of control over struct layout is useful when using platform invocation.

See also

- System.Reflection
- Attribute
- Attributes

Generics and Attributes

Attributes can be applied to generic types in the same way as nongeneric types. However, you can apply attributes only on *open generic types* and *closed constructed generic types*, not on *partially constructed generic types*. An *open generic type* is one where none of the type arguments are specified, such as `Dictionary<TKey, TValue>` A *closed constructed generic type* specifies all type arguments, such as `Dictionary<string, object>`. A *partially constructed generic type* specifies some, but not all, type arguments. An example is `Dictionary<string, TValue>`. An *unbound generic type* is one where type arguments are omitted, such as `Dictionary<,>`.

The following examples use this custom attribute:

C#

```
class CustomAttribute : Attribute
{
    public object? info;
}
```

An attribute can reference an unbound generic type:

C#

```
public class GenericClass1<T> { }

[CustomAttribute(info = typeof(GenericClass1<>))]
class ClassA { }
```

Specify multiple type parameters using the appropriate number of commas. In this example, `GenericClass2` has two type parameters:

C#

```
public class GenericClass2<T, U> { }

[CustomAttribute(info = typeof(GenericClass2<,>))]
class ClassB { }
```

An attribute can reference a closed constructed generic type:

C#

```
public class GenericClass3<T, U, V> { }
```

```
[CustomAttribute(info = typeof(GenericClass3<int, double, string>))]
class ClassC { }
```

An attribute that references a generic type parameter causes a compile-time error:

C#

```
[CustomAttribute(info = typeof(GenericClass3<int, T, string>))]  //Error CS0416
class ClassD<T> { }
```

A generic type can inherit from Attribute:

C#

```
public class CustomGenericAttribute<T> : Attribute { }
```

To obtain information about a generic type or type parameter at run time, you can use the methods of System.Reflection. For more information, see Generics and Reflection.

See also

- Generics
- Attributes

Last updated on 11/18/2025

How to query an assembly's metadata with Reflection (LINQ)

Article • 03/15/2023

You use the .NET reflection APIs to examine the metadata in a .NET assembly and create collections of types, type members, and parameters that are in that assembly. Because these collections support the generic IEnumerable<T> interface, they can be queried by using LINQ.

The following example shows how LINQ can be used with reflection to retrieve specific metadata about methods that match a specified search criterion. In this case, the query finds the names of all the methods in the assembly that return enumerable types such as arrays.

C#

```
Assembly assembly = Assembly.Load("System.Private.CoreLib, Version=7.0.0.0,
Culture=neutral, PublicKeyToken=7cec85d7bea7798e");
var pubTypesQuery = from type in assembly.GetTypes()
                    where type.IsPublic
                    from method in type.GetMethods()
                    where method.ReturnType.IsArray == true
                        || (method.ReturnType.GetInterface(

typeof(System.Collections.Generic.IEnumerable<>).FullName!) != null
                        && method.ReturnType.FullName != "System.String")
                    group method.ToString() by type.ToString();

foreach (var groupOfMethods in pubTypesQuery)
{
    Console.WriteLine($"Type: {groupOfMethods.Key}");
    foreach (var method in groupOfMethods)
    {
        Console.WriteLine($"  {method}");
    }
}
```

The example uses the Assembly.GetTypes method to return an array of types in the specified assembly. The where filter is applied so that only public types are returned. For each public type, a subquery is generated by using the MethodInfo array that is returned from the Type.GetMethods call. These results are filtered to return only those methods whose return type is an array or else a type that implements IEnumerable<T>. Finally, these results are grouped by using the type name as a key.

Generics and reflection

Article • 03/15/2023

Because the Common Language Runtime (CLR) has access to generic type information at run time, you can use reflection to obtain information about generic types in the same way as for nongeneric types. For more information, see Generics in the Runtime.

The System.Reflection.Emit namespace also contains new members that support generics. See How to: Define a Generic Type with Reflection Emit.

For a list of the invariant conditions for terms used in generic reflection, see the IsGenericType property remarks:

- IsGenericType: Returns true if a type is generic.
- GetGenericArguments: Returns an array of `Type` objects that represent the type arguments supplied for a constructed type, or the type parameters of a generic type definition.
- GetGenericTypeDefinition: Returns the underlying generic type definition for the current constructed type.
- GetGenericParameterConstraints: Returns an array of `Type` objects that represent the constraints on the current generic type parameter.
- ContainsGenericParameters: Returns true if the type or any of its enclosing types or methods contain type parameters for which specific types haven't been supplied.
- GenericParameterAttributes: Gets a combination of `GenericParameterAttributes` flags that describe the special constraints of the current generic type parameter.
- GenericParameterPosition: For a `Type` object that represents a type parameter, gets the position of the type parameter in the type parameter list of the generic type definition or generic method definition that declared the type parameter.
- IsGenericParameter: Gets a value that indicates whether the current `Type` represents a type parameter of a generic type or method definition.
- IsGenericTypeDefinition: Gets a value that indicates whether the current Type represents a generic type definition, from which other generic types can be constructed. Returns true if the type represents the definition of a generic type.
- DeclaringMethod: Returns the generic method that defined the current generic type parameter, or null if the type parameter wasn't defined by a generic method.
- MakeGenericType: Substitutes the elements of an array of types for the type parameters of the current generic type definition, and returns a Type object representing the resulting constructed type.

In addition, members of the MethodInfo class enable run-time information for generic methods. See the IsGenericMethod property remarks for a list of invariant conditions for

terms used to reflect on generic methods:

- IsGenericMethod: Returns true if a method is generic.
- GetGenericArguments: Returns an array of Type objects that represent the type arguments of a constructed generic method or the type parameters of a generic method definition.
- GetGenericMethodDefinition: Returns the underlying generic method definition for the current constructed method.
- ContainsGenericParameters: Returns true if the method or any of its enclosing types contain any type parameters for which specific types haven't been supplied.
- IsGenericMethodDefinition: Returns true if the current MethodInfo represents the definition of a generic method.
- MakeGenericMethod: Substitutes the elements of an array of types for the type parameters of the current generic method definition, and returns a MethodInfo object representing the resulting constructed method.

See also

- Generics
- Reflection and Generic Types
- Generics

Define and read custom attributes

Article • 03/15/2023

Attributes provide a way of associating information with code in a declarative way. They can also provide a reusable element that can be applied to various targets. Consider the ObsoleteAttribute. It can be applied to classes, structs, methods, constructors, and more. It *declares* that the element is obsolete. It's then up to the C# compiler to look for this attribute, and do some action in response.

In this tutorial, you learn how to add attributes to your code, how to create and use your own attributes, and how to use some attributes that are built into .NET.

Prerequisites

You need to set up your machine to run .NET. You can find the installation instructions on the .NET Downloads page. You can run this application on Windows, Ubuntu Linux, macOS, or in a Docker container. You need to install your favorite code editor. The following descriptions use Visual Studio Code, which is an open-source, cross-platform editor. However, you can use whatever tools you're comfortable with.

Create the app

Now that you've installed all the tools, create a new .NET console app. To use the command line generator, execute the following command in your favorite shell:

.NET CLI

```
dotnet new console
```

This command creates bare-bones .NET project files. You run `dotnet restore` to restore the dependencies needed to compile this project.

You don't have to run dotnet restore because it's run implicitly by all commands that require a restore to occur, such as `dotnet new`, `dotnet build`, `dotnet run`, `dotnet test`, `dotnet publish`, and `dotnet pack`. To disable implicit restore, use the `--no-restore` option.

The `dotnet restore` command is still useful in certain scenarios where explicitly restoring makes sense, such as continuous integration builds in Azure DevOps Services or in build systems that need to explicitly control when the restore occurs.

For information about how to manage NuGet feeds, see the dotnet restore documentation.

To execute the program, use `dotnet run`. You should see "Hello, World" output to the console.

Add attributes to code

In C#, attributes are classes that inherit from the `Attribute` base class. Any class that inherits from `Attribute` can be used as a sort of "tag" on other pieces of code. For instance, there's an attribute called `ObsoleteAttribute`. This attribute signals that code is obsolete and shouldn't be used anymore. You place this attribute on a class, for instance, by using square brackets.

C#

```
[Obsolete]
public class MyClass
{
}
```

While the class is called `ObsoleteAttribute`, it's only necessary to use `[Obsolete]` in the code. Most C# code follows this convention. You can use the full name `[ObsoleteAttribute]` if you choose.

When marking a class obsolete, it's a good idea to provide some information as to *why* it's obsolete, and/or *what* to use instead. You include a string parameter to the Obsolete attribute to provide this explanation.

C#

```
[Obsolete("ThisClass is obsolete. Use ThisClass2 instead.")]
public class ThisClass
{
}
```

The string is being passed as an argument to an `ObsoleteAttribute` constructor, as if you were writing `var attr = new ObsoleteAttribute("some string")`.

Parameters to an attribute constructor are limited to simple types/literals: `bool, int, double, string, Type, enums, etc` and arrays of those types. You can't use an expression or a variable. You're free to use positional or named parameters.

Create your own attribute

You create an attribute by defining a new class that inherits from the `Attribute` base class.

C#

```
public class MySpecialAttribute : Attribute
{
}
```

With the preceding code, you can use `[MySpecial]` (or `[MySpecialAttribute]`) as an attribute elsewhere in the code base.

C#

```
[MySpecial]
public class SomeOtherClass
{
}
```

Attributes in the .NET base class library like `ObsoleteAttribute` trigger certain behaviors within the compiler. However, any attribute you create acts only as metadata, and doesn't result in any code within the attribute class being executed. It's up to you to act on that metadata elsewhere in your code.

There's a 'gotcha' here to watch out for. As mentioned earlier, only certain types can be passed as arguments when using attributes. However, when creating an attribute type, the C# compiler doesn't stop you from creating those parameters. In the following example, you've created an attribute with a constructor that compiles correctly.

C#

```
public class GotchaAttribute : Attribute
{
    public GotchaAttribute(Foo myClass, string str)
    {
    }
}
```

However, you're unable to use this constructor with attribute syntax.

C#

```
[Gotcha(new Foo(), "test")] // does not compile
public class AttributeFail
{
}
```

The preceding code causes a compiler error like `Attribute constructor parameter 'myClass' has type 'Foo', which is not a valid attribute parameter type`

How to restrict attribute usage

Attributes can be used on the following "targets". The preceding examples show them on classes, but they can also be used on:

- Assembly
- Class
- Constructor
- Delegate
- Enum
- Event
- Field
- GenericParameter
- Interface
- Method
- Module
- Parameter
- Property
- ReturnValue
- Struct

When you create an attribute class, by default, C# allows you to use that attribute on any of the possible attribute targets. If you want to restrict your attribute to certain targets, you can do so by using the `AttributeUsageAttribute` on your attribute class. That's right, an attribute on an attribute!

C#

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class MyAttributeForClassAndStructOnly : Attribute
{
}
```

If you attempt to put the above attribute on something that's not a class or a struct, you get a compiler error like `Attribute 'MyAttributeForClassAndStructOnly' is not valid on this declaration type. It is only valid on 'class, struct' declarations`

C#

```
public class Foo
{
    // if the below attribute was uncommented, it would cause a compiler
error
    // [MyAttributeForClassAndStructOnly]
    public Foo()
    { }
}
```

How to use attributes attached to a code element

Attributes act as metadata. Without some outward force, they don't actually do anything.

To find and act on attributes, reflection is needed. Reflection allows you to write code in C# that examines other code. For instance, you can use Reflection to get information about a class(add `using System.Reflection;` at the head of your code):

C#

```
TypeInfo typeInfo = typeof(MyClass).GetTypeInfo();
Console.WriteLine("The assembly qualified name of MyClass is " +
typeInfo.AssemblyQualifiedName);
```

That prints something like: `The assembly qualified name of MyClass is ConsoleApplication.MyClass, attributes, Version=1.0.0.0, Culture=neutral, PublicKeyToken=null`

Once you have a `TypeInfo` object (or a `MemberInfo`, `FieldInfo`, or other object), you can use the `GetCustomAttributes` method. This method returns a collection of `Attribute` objects. You can also use `GetCustomAttribute` and specify an Attribute type.

Here's an example of using `GetCustomAttributes` on a `MemberInfo` instance for `MyClass` (which we saw earlier has an `[Obsolete]` attribute on it).

C#

```
var attrs = typeInfo.GetCustomAttributes();
foreach(var attr in attrs)
    Console.WriteLine("Attribute on MyClass: " + attr.GetType().Name);
```

That prints to console: `Attribute on MyClass: ObsoleteAttribute`. Try adding other attributes to `MyClass`.

It's important to note that these `Attribute` objects are instantiated lazily. That is, they aren't be instantiated until you use `GetCustomAttribute` or `GetCustomAttributes`. They're also instantiated each time. Calling `GetCustomAttributes` twice in a row returns two different instances of `ObsoleteAttribute`.

Common attributes in the runtime

Attributes are used by many tools and frameworks. NUnit uses attributes like `[Test]` and `[TestFixture]` that are used by the NUnit test runner. ASP.NET MVC uses attributes like `[Authorize]` and provides an action filter framework to perform cross-cutting concerns on MVC actions. PostSharp uses the attribute syntax to allow aspect-oriented programming in C#.

Here are a few notable attributes built into the .NET Core base class libraries:

- `[Obsolete]`. This one was used in the above examples, and it lives in the `System` namespace. It's useful to provide declarative documentation about a changing code base. A message can be provided in the form of a string, and another boolean parameter can be used to escalate from a compiler warning to a compiler error.
- `[Conditional]`. This attribute is in the `System.Diagnostics` namespace. This attribute can be applied to methods (or attribute classes). You must pass a string to the constructor. If that string doesn't match a `#define` directive, then the C# compiler removes any calls to that method (but not the method itself). Typically you use this technique for debugging (diagnostics) purposes.
- `[CallerMemberName]`. This attribute can be used on parameters, and lives in the `System.Runtime.CompilerServices` namespace. `CallerMemberName` is an attribute that is used to inject the name of the method that is calling another method. It's a way to eliminate 'magic strings' when implementing INotifyPropertyChanged in various UI frameworks. As an example:

```
C#
```

```
public class MyUIClass : INotifyPropertyChanged
{
    public event PropertyChangedEventHandler? PropertyChanged;

    public void RaisePropertyChanged([CallerMemberName] string propertyName
= default!)
    {
        PropertyChanged?.Invoke(this, new
PropertyChangedEventArgs(propertyName));
    }

    private string? _name;
    public string? Name
    {
        get { return _name;}
        set
        {
            if (value != _name)
            {
                _name = value;
                RaisePropertyChanged();   // notice that "Name" is not
needed here explicitly
            }
        }
    }
}
```

In the above code, you don't have to have a literal `"Name"` string. Using `CallerMemberName` prevents typo-related bugs and also makes for smoother refactoring/renaming. Attributes bring declarative power to C#, but they're a meta-data form of code and don't act by themselves.

Tutorial: Update interfaces with default interface methods

07/16/2025

You can define an implementation when you declare a member of an interface. The most common scenario is to safely add members to an interface already released and used by innumerable clients.

In this tutorial, you'll learn how to:

- ✓ Extend interfaces safely by adding methods with implementations.
- ✓ Create parameterized implementations to provide greater flexibility.
- ✓ Enable implementers to provide a more specific implementation in the form of an override.

Prerequisites

You need to set up your machine to run .NET, including the C# compiler. The C# compiler is available with Visual Studio 2022 or the .NET SDK.

Scenario overview

This tutorial starts with version 1 of a customer relationship library. You can get the starter application on our samples repo on GitHub. The company that built this library intended customers with existing applications to adopt their library. They provided minimal interface definitions for users of their library to implement. Here's the interface definition for a customer:

C#

```
public interface ICustomer
{
    IEnumerable<IOrder> PreviousOrders { get; }

    DateTime DateJoined { get; }
    DateTime? LastOrder { get; }
    string Name { get; }
    IDictionary<DateTime, string> Reminders { get; }
}
```

They defined a second interface that represents an order:

C#

```
public interface IOrder
{
    DateTime Purchased { get; }
    decimal Cost { get; }
}
```

From those interfaces, the team could build a library for their users to create a better experience for their customers. Their goal was to create a deeper relationship with existing customers and improve their relationships with new customers.

Now, it's time to upgrade the library for the next release. One of the requested features enables a loyalty discount for customers that have lots of orders. This new loyalty discount gets applied whenever a customer makes an order. The specific discount is a property of each individual customer. Each implementation of `ICustomer` can set different rules for the loyalty discount.

The most natural way to add this functionality is to enhance the `ICustomer` interface with a method to apply any loyalty discount. This design suggestion caused concern among experienced developers: "Interfaces are immutable once they've been released! Don't make a breaking change!" You should use default interface implementations for upgrading interfaces. The library authors can add new members to the interface and provide a default implementation for those members.

Default interface implementations enable developers to upgrade an interface while still enabling any implementors to override that implementation. Users of the library can accept the default implementation as a non-breaking change. If their business rules are different, they can override.

Upgrade with default interface methods

The team agreed on the most likely default implementation: a loyalty discount for customers.

The upgrade should provide the functionality to set two properties: the number of orders needed to be eligible for the discount, and the percentage of the discount. These features make it a perfect scenario for default interface methods. You can add a method to the `ICustomer` interface, and provide the most likely implementation. All existing, and any new implementations can use the default implementation, or provide their own.

First, add the new method to the interface, including the body of the method:

```
C#
```

```
// Version 1:
public decimal ComputeLoyaltyDiscount()
{
    DateTime TwoYearsAgo = DateTime.Now.AddYears(-2);
    if ((DateJoined < TwoYearsAgo) && (PreviousOrders.Count() > 10))
    {
        return 0.10m;
    }
    return 0;
}
```

> **Note**
>
> The preceding example uses `DateTime.Now.AddYears(-2)` for simplicity in this tutorial. Be aware that `DateTime` calculations can have edge cases with daylight saving time transitions and leap years. For production code, consider using UTC time or more robust date calculation approaches when precision is important.

The library author wrote a first test to check the implementation:

C#

```
SampleCustomer c = new SampleCustomer("customer one", new DateTime(2010, 5, 31))
{
    Reminders =
    {
        { new DateTime(2010, 08, 12), "childs's birthday" },
        { new DateTime(2012, 11, 15), "anniversary" }
    }
};

SampleOrder o = new SampleOrder(new DateTime(2012, 6, 1), 5m);
c.AddOrder(o);

o = new SampleOrder(new DateTime(2013, 7, 4), 25m);
c.AddOrder(o);

// Check the discount:
ICustomer theCustomer = c;
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Notice the following portion of the test:

C#

```
// Check the discount:
ICustomer theCustomer = c;
```

```
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

That implicit conversion from `SampleCustomer` to `ICustomer` is necessary. The `SampleCustomer` class doesn't need to provide an implementation for `ComputeLoyaltyDiscount`; that's provided by the `ICustomer` interface. However, the `SampleCustomer` class doesn't inherit members from its interfaces. That rule hasn't changed. In order to call any method declared and implemented in the interface, the variable must be the type of the interface, `ICustomer` in this example.

Provide parameterization

The default implementation is too restrictive. Many consumers of this system may choose different thresholds for number of purchases, a different length of membership, or a different percentage discount. You can provide a better upgrade experience for more customers by providing a way to set those parameters. Let's add a static method that sets those three parameters controlling the default implementation:

C#

```
// Version 2:
public static void SetLoyaltyThresholds(
    TimeSpan ago,
    int minimumOrders = 10,
    decimal percentageDiscount = 0.10m)
{
    length = ago;
    orderCount = minimumOrders;
    discountPercent = percentageDiscount;
}
private static TimeSpan length = new TimeSpan(365 * 2, 0,0,0); // two years
private static int orderCount = 10;
private static decimal discountPercent = 0.10m;

public decimal ComputeLoyaltyDiscount()
{
    DateTime start = DateTime.Now - length;

    if ((DateJoined < start) && (PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

There are many new language capabilities shown in that small code fragment. Interfaces can now include static members, including fields and methods. Different access modifiers are also

enabled. The other fields are private, the new method is public. Any of the modifiers are allowed on interface members.

Applications that use the general formula for computing the loyalty discount, but different parameters, don't need to provide a custom implementation; they can set the arguments through a static method. For example, the following code sets a "customer appreciation" that rewards any customer with more than one month's membership:

C#

```
ICustomer.SetLoyaltyThresholds(new TimeSpan(30, 0, 0, 0), 1, 0.25m);
Console.WriteLine($"Current discount: {theCustomer.ComputeLoyaltyDiscount()}");
```

Extend the default implementation

The code you've added so far has provided a convenient implementation for those scenarios where users want something like the default implementation, or to provide an unrelated set of rules. For a final feature, let's refactor the code a bit to enable scenarios where users may want to build on the default implementation.

Consider a startup that wants to attract new customers. They offer a 50% discount off a new customer's first order. Otherwise, existing customers get the standard discount. The library author needs to move the default implementation into a `protected static` method so that any class implementing this interface can reuse the code in their implementation. The default implementation of the interface member calls this shared method as well:

C#

```
public decimal ComputeLoyaltyDiscount() => DefaultLoyaltyDiscount(this);
protected static decimal DefaultLoyaltyDiscount(ICustomer c)
{
    DateTime start = DateTime.Now - length;

    if ((c.DateJoined < start) && (c.PreviousOrders.Count() > orderCount))
    {
        return discountPercent;
    }
    return 0;
}
```

In an implementation of a class that implements this interface, the override can call the static helper method, and extend that logic to provide the "new customer" discount:

C#

```
public decimal ComputeLoyaltyDiscount()
{
   if (PreviousOrders.Any() == false)
        return 0.50m;
    else
        return ICustomer.DefaultLoyaltyDiscount(this);
}
```

You can see the entire finished code in our samples repo on GitHub. You can get the starter application on our samples repo on GitHub.

These new features mean that interfaces can be updated safely when there's a reasonable default implementation for those new members. Carefully design interfaces to express single functional ideas implemented by multiple classes. That makes it easier to upgrade those interface definitions when new requirements are discovered for that same functional idea.

Tutorial: Mix functionality in when creating classes using interfaces with default interface methods

You can define an implementation when you declare a member of an interface. This feature provides new capabilities where you can define default implementations for features declared in interfaces. Classes can pick when to override functionality, when to use the default functionality, and when not to declare support for discrete features.

In this tutorial, you learn how to:

- ✓ Create interfaces with implementations that describe discrete features.
- ✓ Create classes that use the default implementations.
- ✓ Create classes that override some or all of the default implementations.

Prerequisites

You need to set up your machine to run .NET, including the C# compiler. The C# compiler is available with Visual Studio 2022, or the .NET SDK.

Limitations of extensions

One way you can implement behavior that appears as part of an interface is to define extension members that provide the default behavior. Interfaces declare a minimum set of members while providing a greater surface area for any class that implements that interface. For example, the extension members in Enumerable provide the implementation for any sequence to be the source of a LINQ query.

Extension members are resolved at compile time, using the declared type of the variable. Classes that implement the interface can provide a better implementation for any extension member. Variable declarations must match the implementing type to enable the compiler to choose that implementation. When the compile-time type matches the interface, method calls resolve to the extension member. Another concern with extension members is that those members are accessible wherever the class containing the extension member is accessible. Classes can't declare if they should or shouldn't provide features declared in extension members.

You can declare the default implementations as interface methods. Then, every class automatically uses the default implementation. Any class that can provide a better

implementation can override the interface method definition with a better algorithm. In one sense, this technique sounds similar to how you could use extension members.

In this article, you learn how default interface implementations enable new scenarios.

Design the application

Consider a home automation application. You probably have many different types of lights and indicators that could be used throughout the house. Every light must support APIs to turn them on and off, and to report the current state. Some lights and indicators might support other features, such as:

- Turn light on, then turn it off after a timer.
- Blink the light for a period of time.

Some of these extended capabilities could be emulated in devices that support the minimal set. That indicates providing a default implementation. For those devices that have more capabilities built in, the device software would use the native capabilities. For other lights, they could choose to implement the interface and use the default implementation.

Default interface members provide a better solution for this scenario than extension members. Class authors can control which interfaces they choose to implement. Those interfaces they choose are available as methods. In addition, because default interface methods are virtual by default, the method dispatch always chooses the implementation in the class.

Let's create the code to demonstrate these differences.

Create interfaces

Start by creating the interface that defines the behavior for all lights:

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
}
```

A basic overhead light fixture might implement this interface as shown in the following code:

C#

```
public class OverheadLight : ILight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

In this tutorial, the code doesn't drive IoT devices, but emulates those activities by writing messages to the console. You can explore the code without automating your house.

Next, let's define the interface for a light that can automatically turn off after a timeout:

C#

```
public interface ITimerLight : ILight
{
    Task TurnOnFor(int duration);
}
```

You could add a basic implementation to the overhead light, but a better solution is to modify this interface definition to provide a `virtual` default implementation:

C#

```
public interface ITimerLight : ILight
{
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Using the default interface method for the
ITimerLight.TurnOnFor.");
        SwitchOn();
        await Task.Delay(duration);
        SwitchOff();
        Console.WriteLine("Completed ITimerLight.TurnOnFor sequence.");
    }
}
```

The `OverheadLight` class can implement the timer function by declaring support for the interface:

C#

```
public class OverheadLight : ITimerLight { }
```

A different light type might support a more sophisticated protocol. It can provide its own implementation for `TurnOnFor`, as shown in the following code:

C#

```
public class HalogenLight : ITimerLight
{
    private enum HalogenLightState
    {
        Off,
        On,
        TimerModeOn
    }

    private HalogenLightState state;
    public void SwitchOn() => state = HalogenLightState.On;
    public void SwitchOff() => state = HalogenLightState.Off;
    public bool IsOn() => state != HalogenLightState.Off;
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Halogen light starting timer function.");
        state = HalogenLightState.TimerModeOn;
        await Task.Delay(duration);
        state = HalogenLightState.Off;
        Console.WriteLine("Halogen light finished custom timer function");
    }

    public override string ToString() => $"The light is {state}";
}
```

Unlike overriding virtual class methods, the declaration of `TurnOnFor` in the `HalogenLight` class doesn't use the `override` keyword.

Mix and match capabilities

The advantages of default interface methods become clearer as you introduce more advanced capabilities. Using interfaces enables you to mix and match capabilities. It also enables each class author to choose between the default implementation and a custom implementation. Let's add an interface with a default implementation for a blinking light:

C#

```
public interface IBlinkingLight : ILight
{
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Using the default interface method for
IBlinkingLight.Blink.");
        for (int count = 0; count < repeatCount; count++)
```

```
        {
            SwitchOn();
            await Task.Delay(duration);
            SwitchOff();
            await Task.Delay(duration);
        }
        Console.WriteLine("Done with the default interface method for
IBlinkingLight.Blink.");
    }
}
```

The default implementation enables any light to blink. The overhead light can add both timer and blink capabilities using the default implementation:

C#

```
public class OverheadLight : ILight, ITimerLight, IBlinkingLight
{
    private bool isOn;
    public bool IsOn() => isOn;
    public void SwitchOff() => isOn = false;
    public void SwitchOn() => isOn = true;

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

A new light type, the `LEDLight` supports both the timer function and the blink function directly. This light style implements both the `ITimerLight` and `IBlinkingLight` interfaces, and overrides the `Blink` method:

C#

```
public class LEDLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("LED Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("LED Light has finished the Blink function.");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

An `ExtraFancyLight` might support both blink and timer functions directly:

C#

```
public class ExtraFancyLight : IBlinkingLight, ITimerLight, ILight
{
    private bool isOn;
    public void SwitchOn() => isOn = true;
    public void SwitchOff() => isOn = false;
    public bool IsOn() => isOn;
    public async Task Blink(int duration, int repeatCount)
    {
        Console.WriteLine("Extra Fancy Light starting the Blink function.");
        await Task.Delay(duration * repeatCount);
        Console.WriteLine("Extra Fancy Light has finished the Blink function.");
    }
    public async Task TurnOnFor(int duration)
    {
        Console.WriteLine("Extra Fancy light starting timer function.");
        await Task.Delay(duration);
        Console.WriteLine("Extra Fancy light finished custom timer function");
    }

    public override string ToString() => $"The light is {(isOn ? "on" : "off")}";
}
```

The `HalogenLight` you created earlier doesn't support blinking. So, don't add the `IBlinkingLight` to the list of its supported interfaces.

Detect the light types using pattern matching

Next, let's write some test code. You can make use of C#'s pattern matching feature to determine a light's capabilities by examining which interfaces it supports. The following method exercises the supported capabilities of each light:

C#

```
private static async Task TestLightCapabilities(ILight light)
{
    // Perform basic tests:
    light.SwitchOn();
    Console.WriteLine($"\tAfter switching on, the light is {(light.IsOn() ? "on" :
"off")}");
    light.SwitchOff();
    Console.WriteLine($"\tAfter switching off, the light is {(light.IsOn() ? "on" :
"off")}");

    if (light is ITimerLight timer)
    {
        Console.WriteLine("\tTesting timer function");
        await timer.TurnOnFor(1000);
        Console.WriteLine("\tTimer function completed");
    }
```

```
    else
    {
        Console.WriteLine("\tTimer function not supported.");
    }

    if (light is IBlinkingLight blinker)
    {
        Console.WriteLine("\tTesting blinking function");
        await blinker.Blink(500, 5);
        Console.WriteLine("\tBlink function completed");
    }
    else
    {
        Console.WriteLine("\tBlink function not supported.");
    }
}
```

The following code in your `Main` method creates each light type in sequence and tests that light:

C#

```
static async Task Main(string[] args)
{
    Console.WriteLine("Testing the overhead light");
    var overhead = new OverheadLight();
    await TestLightCapabilities(overhead);
    Console.WriteLine();

    Console.WriteLine("Testing the halogen light");
    var halogen = new HalogenLight();
    await TestLightCapabilities(halogen);
    Console.WriteLine();

    Console.WriteLine("Testing the LED light");
    var led = new LEDLight();
    await TestLightCapabilities(led);
    Console.WriteLine();

    Console.WriteLine("Testing the fancy light");
    var fancy = new ExtraFancyLight();
    await TestLightCapabilities(fancy);
    Console.WriteLine();
}
```

How the compiler determines best implementation

This scenario shows a base interface without any implementations. Adding a method into the `ILight` interface introduces new complexities. The language rules governing default interface methods minimize the effect on the concrete classes that implement multiple derived

interfaces. Let's enhance the original interface with a new method to show how that changes its use. Every indicator light can report its power status as an enumerated value:

C#

```
public enum PowerStatus
{
    NoPower,
    ACPower,
    FullBattery,
    MidBattery,
    LowBattery
}
```

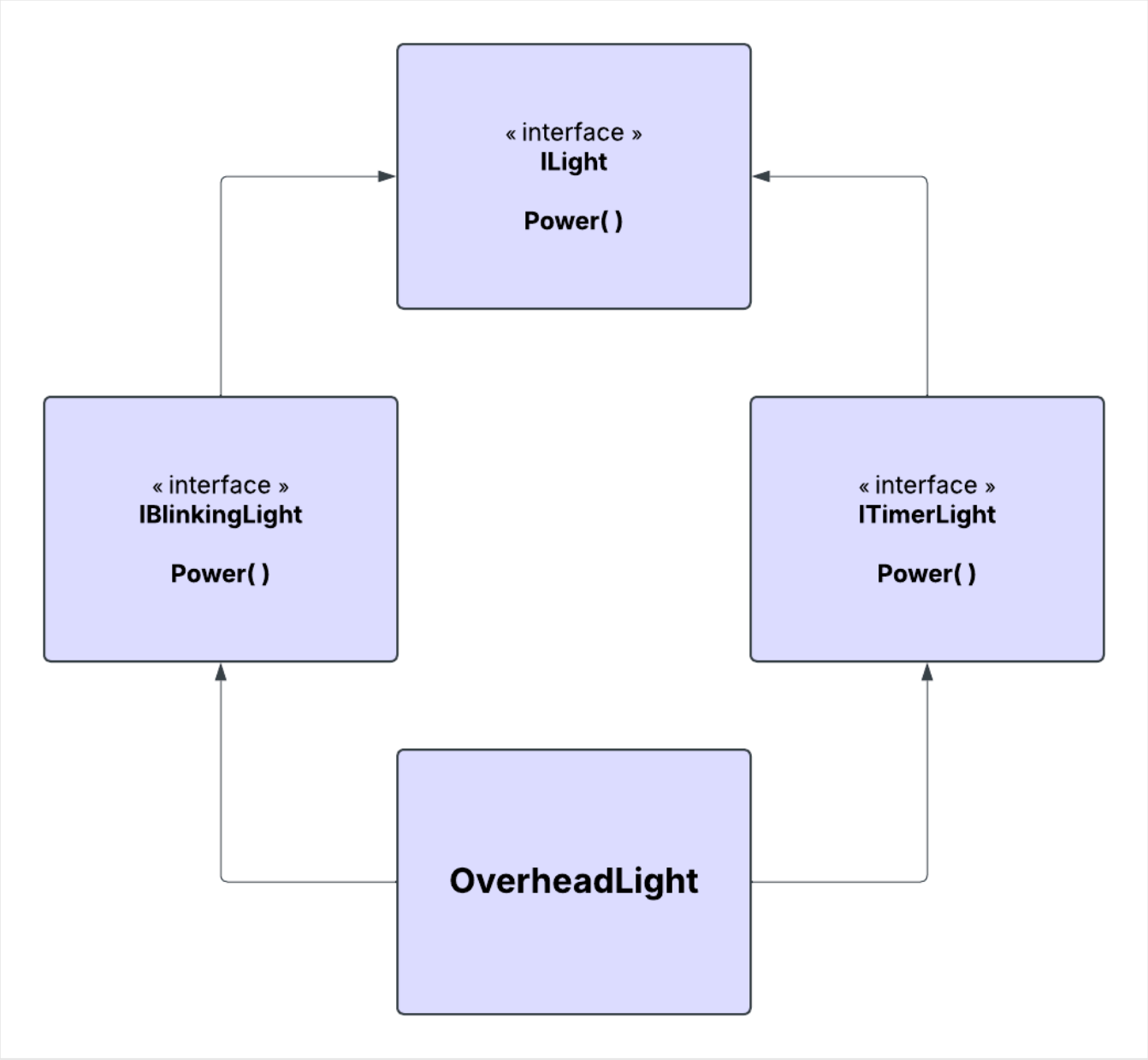
The default implementation assumes no power:

C#

```
public interface ILight
{
    void SwitchOn();
    void SwitchOff();
    bool IsOn();
    public PowerStatus Power() => PowerStatus.NoPower;
}
```

These changes compile cleanly, even though the `ExtraFancyLight` declares support for the `ILight` interface and both derived interfaces, `ITimerLight` and `IBlinkingLight`. There's only one "closest" implementation declared in the `ILight` interface. Any class that declared an override would become the one "closest" implementation. You saw examples in the preceding classes that overrode the members of other derived interfaces.

Avoid overriding the same method in multiple derived interfaces. Doing so creates an ambiguous method call whenever a class implements both derived interfaces. The compiler can't pick a single better method so it issues an error. For example, if both the `IBlinkingLight` and `ITimerLight` implemented an override of `Power()`, the `OverheadLight` would need to provide a more specific override. Otherwise, the compiler can't pick between the implementations in the two derived interfaces. This situation is shown in the following diagram:



The preceding diagram illustrates the ambiguity. `OverheadLight` doesn't provide an implementation of `ILight.Power()`. Both `IBlinkingLight` and `ITimerLight` provide overrides that are more specific. A call to `ILight.Power()` on an instance of `OverheadLight` is ambiguous. You must add a new override in `OverheadLight` to resolve the ambiguity.

You can usually avoid this situation by keeping interface definitions small and focused on one feature. In this scenario, each capability of a light is its own interface; only classes inherit multiple interfaces.

This sample shows one scenario where you can define discrete features that can be mixed into classes. You declare any set of supported functionality by declaring which interfaces a class supports. The use of virtual default interface methods enables classes to use or define a different implementation for any or all the interface methods. This language capability provides new ways to model the real-world systems you're building. Default interface methods provide a clearer way to express related classes that might mix and match different features using virtual implementations of those capabilities.

Last updated on 11/26/2025

Tutorial: Explore static virtual members in interfaces

Interface static virtual members enable you to define interfaces that include overloaded operators or other static members. Once you define interfaces with static members, you can use those interfaces as constraints to create generic types that use operators or other static methods. Even if you don't create interfaces with overloaded operators, you likely benefit from this feature and the generic math classes enabled by the language update.

In this tutorial, you learn how to:

- ✓ Define interfaces with static abstract and static virtual members.
- ✓ Use interfaces to define classes that implement interfaces with operators defined.
- ✓ Provide default implementations with static virtual methods.
- ✓ Create generic algorithms that rely on static interface methods.

Prerequisites

- The latest .NET SDK
- Visual Studio Code editor
- The C# DevKit

Static abstract interface methods

Let's start with an example. The following method returns the midpoint of two `double` numbers:

C#

```
public static double MidPoint(double left, double right) =>
    (left + right) / (2.0);
```

The same logic would work for any numeric type: `int`, `short`, `long`, `float` `decimal`, or any type that represents a number. You need to have a way to use the `+` and `/` operators, and to define a value for `2`. You can use the System.Numerics.INumber<TSelf> interface to write the preceding method as the following generic method:

C#

```
public static T MidPoint<T>(T left, T right)
    where T : INumber<T> => (left + right) / T.CreateChecked(2);  // note: the
```

```
addition of left and right may overflow here; it's just for demonstration purposes
```

Any type that implements the INumber<TSelf> interface must include a definition for `operator +`, and for `operator /`. The denominator is defined by `T.CreateChecked(2)` to create the value `2` for any numeric type, which forces the denominator to be the same type as the two parameters. INumberBase<TSelf>.CreateChecked<TOther>(TOther) creates an instance of the type from the specified value and throws an OverflowException if the value falls outside the representable range. (This implementation has the potential for overflow if `left` and `right` are both large enough values. There are alternative algorithms that can avoid this potential issue.)

You define static abstract members in an interface using familiar syntax: You add the `static` and `abstract` modifiers to any static member that doesn't provide an implementation. The following example defines an `IGetNext<T>` interface that can be applied to any type that overrides `operator ++`:

C#

```
public interface IGetNext<T> where T : IGetNext<T>
{
    static abstract T operator ++(T other);
}
```

The constraint that the type argument, `T`, implements `IGetNext<T>` ensures that the signature for the operator includes the containing type, or its type argument. Many operators enforce that its parameters must match the type, or be the type parameter constrained to implement the containing type. Without this constraint, the `++` operator couldn't be defined in the `IGetNext<T>` interface.

You can create a structure that creates a string of 'A' characters where each increment adds another character to the string using the following code:

C#

```
public struct RepeatSequence : IGetNext<RepeatSequence>
{
    private const char Ch = 'A';
    public string Text = new string(Ch, 1);

    public RepeatSequence() {}

    public static RepeatSequence operator ++(RepeatSequence other)
        => other with { Text = other.Text + Ch };

    public override string ToString() => Text;
}
```

More generally, you can build any algorithm where you might want to define `++` to mean "produce the next value of this type." Using this interface produces clear code and results:

C#

```
var str = new RepeatSequence();

for (int i = 0; i < 10; i++)
    Console.WriteLine(str++);
```

The preceding example produces the following output:

PowerShell

```
A
AA
AAA
AAAA
AAAAA
AAAAAA
AAAAAAA
AAAAAAAA
AAAAAAAAA
AAAAAAAAAA
```

This small example demonstrates the motivation for this feature. You can use natural syntax for operators, constant values, and other static operations. You can explore these techniques when you create multiple types that rely on static members, including overloaded operators. Define the interfaces that match your types' capabilities and then declare those types' support for the new interface.

Static virtual interface methods

The previous example used `static abstract` to declare the `++` operator: every implementing type *must* provide its own implementation. You can also declare `static virtual` members that provide a default implementation. Implementing types can use the default or override it. This distinction is useful when there's a reasonable default that works for most types, but some types need specialized behavior.

The following example defines an `IDescribable<T>` interface with a `static abstract` property, `TypeName`, and a `static virtual` method, `Describe()`. The `TypeName` property is abstract because each type must supply its own name. The `Describe()` method is virtual because returning the type name is a sensisble default for most types:

C#

```
public interface IDescribable<T> where T : IDescribable<T>
{
    static abstract string TypeName { get; }
    static virtual string Describe() => T.TypeName;
}
```

A type that implements `IDescribable<T>` must provide `TypeName`. It can choose whether to override `Describe()`. The `Widget` type relies on the default:

C#

```
public struct Widget : IDescribable<Widget>
{
    public static string TypeName => "Widget";

    // Uses the default Describe(): returns "Widget"
}
```

The `Gadget` type overrides `Describe()` to include additional detail:

C#

```
public struct Gadget : IDescribable<Gadget>
{
    public static string TypeName => "Gadget";

    // Overrides the default Describe() with a custom message
    public static string Describe() => $"{TypeName} (version 2.0)";
}
```

You call `static virtual` members through the type parameter in a generic method, just like `static abstract` members:

C#

```
static void PrintDescription<T>() where T : IDescribable<T>
{
    Console.WriteLine(T.Describe());
}

PrintDescription<Widget>();
PrintDescription<Gadget>();
```

The preceding example produces the following output:

PowerShell

```
Widget
Gadget (version 2.0)
```

`Widget` uses the default implementation of `Describe()`, which returns `TypeName`. `Gadget` overrides `Describe()` with its own version. The compiler resolves the correct implementation at compile time based on the type argument. In summary, use `static abstract` when every implementing type must provide its own implementation. Use `static virtual` when you can provide a useful default that most types will accept.

Generic math

The motivating scenario for allowing static methods, including operators, in interfaces is to support generic math algorithms. The .NET 7 base class library contains interface definitions for many arithmetic operators, and derived interfaces that combine many arithmetic operators in an `INumber<T>` interface. Let's apply those types to build a `Point<T>` record that can use any numeric type for `T`. You can move the point by some `XOffset` and `YOffset` using the `+` operator.

Start by creating a new Console application, either by using `dotnet new` or Visual Studio.

The public interface for the `Translation<T>` and `Point<T>` should look like the following code:

C#

```
// Note: Not complete. This won't compile yet.
public record Translation<T>(T XOffset, T YOffset);

public record Point<T>(T X, T Y)
{
    public static Point<T> operator +(Point<T> left, Translation<T> right);
}
```

You use the `record` type for both the `Translation<T>` and `Point<T>` types: Both store two values, and they represent data storage rather than sophisticated behavior. The implementation of `operator +` would look like the following code:

C#

```
public static Point<T> operator +(Point<T> left, Translation<T> right) =>
    left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
```

For the previous code to compile, you need to declare that `T` supports the `IAdditionOperators<TSelf, TOther, TResult>` interface. That interface includes the `operator +` static method. It declares three type parameters: One for the left operand, one for the right operand, and one for the result. Some types implement `+` for different operand and result types. Add a declaration that the type argument, `T` implements `IAdditionOperators<T, T, T>`:

C#

```
public record Point<T>(T X, T Y) where T : IAdditionOperators<T, T, T>
```

After you add that constraint, your `Point<T>` class can use the `+` for its addition operator. Add the same constraint on the `Translation<T>` declaration:

C#

```
public record Translation<T>(T XOffset, T YOffset) where T : IAdditionOperators<T,
T, T>;
```

The `IAdditionOperators<T, T, T>` constraint prevents a developer using your class from creating a `Translation` using a type that doesn't meet the constraint for the addition to a point. You added the necessary constraints to the type parameter for `Translation<T>` and `Point<T>` so this code works. You can test by adding code like the following above the declarations of `Translation` and `Point` in your *Program.cs* file:

C#

```
var pt = new Point<int>(3, 4);

var translate = new Translation<int>(5, 10);

var final = pt + translate;

Console.WriteLine(pt);
Console.WriteLine(translate);
Console.WriteLine(final);
```

You can make this code more reusable by declaring that these types implement the appropriate arithmetic interfaces. The first change to make is to declare that `Point<T, T>` implements the `IAdditionOperators<Point<T>, Translation<T>, Point<T>>` interface. The `Point` type makes use of different types for operands and the result. The `Point` type already implements an `operator +` with that signature, so adding the interface to the declaration is all you need:

C#

```
public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>,
Point<T>>
    where T : IAdditionOperators<T, T, T>
```

Finally, when you're performing addition, it's useful to have a property that defines the additive identity value for that type. There's a new interface for that feature: IAdditiveIdentity<TSelf,TResult>. A translation of `{0, 0}` is the additive identity: The resulting point is the same as the left operand. The `IAdditiveIdentity<TSelf, TResult>` interface defines one readonly property, `AdditiveIdentity`, that returns the identity value. The `Translation<T>` needs a few changes to implement this interface:

C#

```
using System.Numerics;

public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Translation<T> AdditiveIdentity =>
        new Translation<T>(XOffset: T.AdditiveIdentity, YOffset:
T.AdditiveIdentity);
}
```

There are a few changes here, so let's walk through them one by one. First, you declare that the `Translation` type implements the `IAdditiveIdentity` interface:

C#

```
public record Translation<T>(T XOffset, T YOffset) :
IAdditiveIdentity<Translation<T>, Translation<T>>
```

You next might try implementing the interface member as shown in the following code:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: 0, YOffset: 0);
```

The preceding code doesn't compile, because `0` depends on the type. The answer: Use `IAdditiveIdentity<T>.AdditiveIdentity` for `0`. That change means that your constraints must now include that `T` implements `IAdditiveIdentity<T>`. That results in the following implementation:

C#

```
public static Translation<T> AdditiveIdentity =>
    new Translation<T>(XOffset: T.AdditiveIdentity, YOffset: T.AdditiveIdentity);
```

Now that you added that constraint on `Translation<T>`, you need to add the same constraint to `Point<T>`:

C#

```
using System.Numerics;

public record Point<T>(T X, T Y) : IAdditionOperators<Point<T>, Translation<T>,
Point<T>>
    where T : IAdditionOperators<T, T, T>, IAdditiveIdentity<T, T>
{
    public static Point<T> operator +(Point<T> left, Translation<T> right) =>
        left with { X = left.X + right.XOffset, Y = left.Y + right.YOffset };
}
```

This sample gave you a look at how the interfaces for generic math compose. You learned how to:

- ✓ Write a method that relied on the `INumber<T>` interface so that method could be used with any numeric type.
- ✓ Build a type that relies on the addition interfaces to implement a type that only supports one mathematical operation. That type declares its support for those same interfaces so it can be composed in other ways. The algorithms are written using the most natural syntax of mathematical operators.

Experiment with these features and register feedback. You can use the *Send Feedback* menu item in Visual Studio, or create a new issue in the roslyn repository on GitHub. Build generic algorithms that work with any numeric type. Build algorithms using these interfaces where the type argument only implements a subset of number-like capabilities. Even if you don't build new interfaces that use these capabilities, you can experiment with using them in your algorithms.

See also

- Generic math

Last updated on 03/13/2026

Expression Trees

10/13/2025

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`.

If you used LINQ, you have experience with a rich library where the `Func` types are part of the API set. (If you aren't familiar with LINQ, you probably want to read the LINQ tutorial and the article about lambda expressions before this one.) Expression Trees provide richer interaction with the arguments that are functions.

You write function arguments, typically using Lambda Expressions, when you create LINQ queries. In a typical LINQ query, those function arguments are transformed into a delegate the compiler creates.

You already write code that uses Expression trees. Entity Framework's LINQ APIs accept Expression trees as the arguments for the LINQ Query Expression Pattern. That enables Entity Framework to translate the query you wrote in C# into SQL that executes in the database engine. Another example is Moq, which is a popular mocking framework for .NET.

When you want to have a richer interaction, you need to use *Expression Trees*. Expression Trees represent code as a structure that you examine, modify, or execute. These tools give you the power to manipulate code during run time. You write code that examines running algorithms, or injects new capabilities. In more advanced scenarios, you modify running algorithms and even translate C# expressions into another form for execution in another environment.

You compile and run code represented by expression trees. Building and running expression trees 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.

Expression trees are also used in the dynamic language runtime (DLR) to provide interoperability between dynamic languages and .NET and to enable compiler writers to emit expression trees instead of Microsoft intermediate language (CIL). 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.

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

```
C#
```

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

You create expression trees in your code. You build the tree by creating each node and attaching the nodes into a tree structure. You learn how to create expressions in the article on building expression trees.

Expression trees are immutable. 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 use an expression tree visitor to traverse the existing expression tree. For more information, see the article on translating expression trees.

Once you build an expression tree, you execute the code represented by the expression tree.

Limitations

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

There are some newer C# language elements that don't translate well into expression trees. Expression trees can't contain `await` expressions, or `async` lambda expressions. Many of the features added in C# 6 and later don't appear exactly as written in expression trees. Instead, newer features are exposed in expression trees in the equivalent, earlier syntax, where possible. Other constructs aren't available. It means that code that interprets expression trees works the same when new language features are introduced. However, even with these limitations, expression trees do enable you to create dynamic algorithms that rely on interpreting and modifying code that is represented as a data structure. It enables rich libraries such as Entity Framework to accomplish what they do.

Expression trees don't support new expression node types. It would be a breaking change for all libraries interpreting expression trees to introduce new node types. The following list includes most C# language elements that can't be used:

- Conditional methods removed from the output
- base access
- Method group expressions, including *address-of* (&) a method group, and anonymous method expressions

- References to local functions
- Statements, including assignment (`=`) and statement bodied expressions
- Partial methods with only a defining declaration
- Unsafe pointer operations
- dynamic operations
- Coalescing operators with null or default literal left side, null coalescing assignment, and the null propagating operator (?.)
- Multi-dimensional array initializers, indexed properties, and dictionary initializers
- Collection expressions
- throw expressions
- Accessing static virtual or abstract interface members
- Lambda expressions that have attributes
- Interpolated strings
- UTF-8 string conversions or UTF-8 string literals
- Method invocations using variable arguments, named arguments, or optional arguments
- Expressions using System.Index or System.Range, index "from end" (^) operator or range expressions (..)
- async lambda expressions or await expressions, including await foreach and await using
- Tuple literals, tuple conversions, tuple == or !=, or with expressions
- Discards (_), deconstructing assignment, pattern matching is operator, or the pattern matching switch expression
- COM call with `ref` omitted on the arguments
- ref, in or out parameters, `ref` return values, `out` arguments, or any values of ref struct type

Expression trees - data that defines code

Article • 03/09/2023

An Expression Tree is a data structure that defines code. Expression trees are based on the same structures that a compiler uses to analyze code and generate the compiled output. As you read this article, you notice quite a bit of similarity between Expression Trees and the types used in the Roslyn APIs to build Analyzers and CodeFixes. (Analyzers and CodeFixes are NuGet packages that perform static analysis on code and suggest potential fixes for a developer.) The concepts are similar, and the end result is a data structure that allows examination of the source code in a meaningful way. However, Expression Trees are based on a different set of classes and APIs than the Roslyn APIs. Here's a line of code:

```
C#

var sum = 1 + 2;
```

If you analyze the preceding code as an expression tree, the tree contains several nodes. The outermost node is a variable declaration statement with assignment (`var sum = 1 + 2;`) That outermost node contains several child nodes: a variable declaration, an assignment operator, and an expression representing the right hand side of the equals sign. That expression is further subdivided into expressions that represent the addition operation, and left and right operands of the addition.

Let's drill down a bit more into the expressions that make up the right side of the equals sign. The expression is `1 + 2`, a binary expression. More specifically, it's a binary addition expression. A binary addition expression has two children, representing the left and right nodes of the addition expression. Here, both nodes are constant expressions: The left operand is the value `1`, and the right operand is the value `2`.

Visually, the entire statement is a tree: You could start at the root node, and travel to each node in the tree to see the code that makes up the statement:

- Variable declaration statement with assignment (`var sum = 1 + 2;`)
  - Implicit variable type declaration (`var sum`)
    - Implicit var keyword (`var`)
    - Variable name declaration (`sum`)
  - Assignment operator (`=`)
  - Binary addition expression (`1 + 2`)

- Left operand (`1`)
- Addition operator (`+`)
- Right operand (`2`)

The preceding tree may look complicated, but it's very powerful. Following the same process, you decompose much more complicated expressions. Consider this expression:

```
C#
```

```
var finalAnswer = this.SecretSauceFunction(
    currentState.createInterimResult(), currentState.createSecondValue(1,
2),
    decisionServer.considerFinalOptions("hello")) +
    MoreSecretSauce('A', DateTime.Now, true);
```

The preceding expression is also a variable declaration with an assignment. In this instance, the right hand side of the assignment is a much more complicated tree. You're not going to decompose this expression, but consider what the different nodes might be. There are method calls using the current object as a receiver, one that has an explicit `this` receiver, one that doesn't. There are method calls using other receiver objects, there are constant arguments of different types. And finally, there's a binary addition operator. Depending on the return type of `SecretSauceFunction()` or `MoreSecretSauce()`, that binary addition operator may be a method call to an overridden addition operator, resolving to a static method call to the binary addition operator defined for a class.

Despite this perceived complexity, the preceding expression creates a tree structure navigated as easily as the first sample. You keep traversing child nodes to find leaf nodes in the expression. Parent nodes have references to their children, and each node has a property that describes what kind of node it is.

The structure of an expression tree is very consistent. Once you've learned the basics, you understand even the most complex code when it's represented as an expression tree. The elegance in the data structure explains how the C# compiler analyzes the most complex C# programs and creates proper output from that complicated source code.

Once you become familiar with the structure of expression trees, you find that knowledge you've gained quickly enables you to work with many more advanced scenarios. There's incredible power to expression trees.

In addition to translating algorithms to execute in other environments, expression trees make it easier to write algorithms that inspect code before executing it. You write a method whose arguments are expressions and then examine those expressions before

executing the code. The Expression Tree is a full representation of the code: you see values of any subexpression. You see method and property names. You see the value of any constant expressions. You convert an expression tree into an executable delegate, and execute the code.

The APIs for Expression Trees enable you to create trees that represent almost any valid code construct. However, to keep things as simple as possible, some C# idioms can't be created in an expression tree. One example is asynchronous expressions (using the `async` and `await` keywords). If your needs require asynchronous algorithms, you would need to manipulate the `Task` objects directly, rather than rely on the compiler support. Another is in creating loops. Typically, you create these loops by using `for`, `foreach`, `while` or `do` loops. As you see later in this series, the APIs for expression trees support a single loop expression, with `break` and `continue` expressions that control repeating the loop.

The one thing you can't do is modify an expression tree. Expression Trees are immutable data structures. If you want to mutate (change) an expression tree, you must create a new tree that is a copy of the original, but with your desired changes.

.NET Runtime support for expression trees

Article • 03/09/2023

There's a large list of classes in the .NET runtime that work with Expression Trees. You can see the full list at System.Linq.Expressions. Rather than enumerate the full list, let's understand how the runtime classes have been designed.

In language design, an expression is a body of code that evaluates and returns a value. Expressions may be simple: the constant expression `1` returns the constant value of 1. They may be more complicated: The expression `(-B + Math.Sqrt(B*B - 4 * A * C)) / (2 * A)` returns one root for a quadratic equation (in the case where the equation has a solution).

System.Linq.Expression and derived types

One of the complexities of working with expression trees is that many different kinds of expressions are valid in many places in programs. Consider an assignment expression. The right hand side of an assignment could be a constant value, a variable, a method call expression, or others. That language flexibility means that you may encounter many different expression types anywhere in the nodes of a tree when you traverse an expression tree. Therefore, when you work with the base expression type, that's the simplest way to work. However, sometimes you need to know more. The base Expression class contains a `NodeType` property for this purpose. It returns an `ExpressionType`, which is an enumeration of possible expression types. Once you know the type of the node, you cast it to that type, and perform specific actions knowing the type of the expression node. You can search for certain node types, and then work with the specific properties of that kind of expression.

For example, this code prints the name of a variable for a variable access expression. The following code shows the practice of checking the node type, then casting to a variable access expression and then checking the properties of the specific expression type:

C#

```
Expression<Func<int, int>> addFive = (num) => num + 5;

if (addFive is LambdaExpression lambdaExp)
{
    var parameter = lambdaExp.Parameters[0];  -- first
```

```
    Console.WriteLine(parameter.Name);
    Console.WriteLine(parameter.Type);
}
```

Create expression trees

The `System.Linq.Expression` class also contains many static methods to create expressions. These methods create an expression node using the arguments supplied for its children. In this way, you build up an expression from its leaf nodes. For example, this code builds an Add expression:

C#

```
// Addition is an add expression for "1 + 2"
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
```

You can see from this simple example that many types are involved in creating and working with expression trees. That complexity is necessary to provide the capabilities of the rich vocabulary provided by the C# language.

Navigate the APIs

There are Expression node types that map to almost all of the syntax elements of the C# language. Each type has specific methods for that type of language element. It's a lot to keep in your head at one time. Rather than try to memorize everything, here are the techniques you use to work with Expression trees:

1. Look at the members of the `ExpressionType` enum to determine possible nodes you should be examining. This list helps when you want to traverse and understand an expression tree.
2. Look at the static members of the `Expression` class to build an expression. Those methods can build any expression type from a set of its child nodes.
3. Look at the `ExpressionVisitor` class to build a modified expression tree.

You find more as you look at each of those three areas. Invariably, you find what you need when you start with one of those three steps.

Execute expression trees

Article • 03/09/2023

An *expression tree* is a data structure that represents some code. It isn't compiled and executable code. If you want to execute the .NET code represented by an expression tree, you must convert it into executable IL instructions. Executing an expression tree may return a value, or it may just perform an action such as calling a method.

Only expression trees that represent lambda expressions can be executed. Expression trees that represent lambda expressions are of type LambdaExpression or Expression<TDelegate>. To execute these expression trees, call the Compile method to create an executable delegate, and then invoke the delegate.

> **Note**
>
> If the type of the delegate is not known, that is, the lambda expression is of type LambdaExpression and not Expression<TDelegate>, call the DynamicInvoke method on the delegate instead of invoking it directly.

If an expression tree doesn't represent a lambda expression, you can create a new lambda expression that has the original expression tree as its body, by calling the Lambda<TDelegate>(Expression, IEnumerable<ParameterExpression>) method. Then, you can execute the lambda expression as described earlier in this section.

Lambda expressions to functions

You can convert any LambdaExpression, or any type derived from LambdaExpression into executable IL. Other expression types can't be directly converted into code. This restriction has little effect in practice. Lambda expressions are the only types of expressions that you would want to execute by converting to executable intermediate language (IL). (Think about what it would mean to directly execute a System.Linq.Expressions.ConstantExpression. Would it mean anything useful?) Any expression tree that is a System.Linq.Expressions.LambdaExpression, or a type derived from `LambdaExpression` can be converted to IL. The expression type System.Linq.Expressions.Expression<TDelegate> is the only concrete example in the .NET Core libraries. It's used to represent an expression that maps to any delegate type. Because this type maps to a delegate type, .NET can examine the expression, and generate IL for an appropriate delegate that matches the signature of the lambda expression. The delegate type is based on the expression type. You must know the

return type and the argument list if you want to use the delegate object in a strongly typed manner. The `LambdaExpression.Compile()` method returns the `Delegate` type. You have to cast it to the correct delegate type to have any compile-time tools check the argument list or return type.

In most cases, a simple mapping between an expression and its corresponding delegate exists. For example, an expression tree represented by `Expression<Func<int>>` would be converted to a delegate of the type `Func<int>`. For a lambda expression with any return type and argument list, there exists a delegate type that is the target type for the executable code represented by that lambda expression.

The System.Linq.Expressions.LambdaExpression type contains LambdaExpression.Compile and LambdaExpression.CompileToMethod members that you would use to convert an expression tree to executable code. The `Compile` method creates a delegate. The `CompileToMethod` method updates a System.Reflection.Emit.MethodBuilder object with the IL that represents the compiled output of the expression tree.

> **ⓘ Important**
>
> `CompileToMethod` is only available in .NET Framework, not in .NET Core or .NET 5 and later.

Optionally, you can also provide a System.Runtime.CompilerServices.DebugInfoGenerator that receives the symbol debugging information for the generated delegate object. The `DebugInfoGenerator` provides full debugging information about the generated delegate.

You would convert an expression into a delegate using the following code:

C#

```
Expression<Func<int>> add = () => 1 + 2;
var func = add.Compile(); // Create Delegate
var answer = func(); // Invoke Delegate
Console.WriteLine(answer);
```

The following code example demonstrates the concrete types used when you compile and execute an expression tree.

C#

```
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.
```

The following code example demonstrates how to execute an expression tree that represents raising a number to a power by creating a lambda expression and executing it. The result, which represents the number raised to the power, is displayed.

C#

```
// The expression tree to execute.
BinaryExpression be = Expression.Power(Expression.Constant(2d),
Expression.Constant(3d));

// Create a lambda expression.
Expression<Func<double>> le = Expression.Lambda<Func<double>>(be);

// Compile the lambda expression.
Func<double> compiledExpression = le.Compile();

// Execute the lambda expression.
double result = compiledExpression();

// Display the result.
Console.WriteLine(result);

// This code produces the following output:
// 8
```

Execution and lifetimes

You execute the code by invoking the delegate created when you called `LambdaExpression.Compile()`. The preceding code, `add.Compile()`, returns a delegate. You invoke that delegate by calling `func()`, which executes the code.

That delegate represents the code in the expression tree. You can retain the handle to that delegate and invoke it later. You don't need to compile the expression tree each time you want to execute the code it represents. (Remember that expression trees are immutable, and compiling the same expression tree later creates a delegate that executes the same code.)

> **⊗ Caution**
>
> Don't create any more sophisticated caching mechanisms to increase performance by avoiding unnecessary compile calls. Comparing two arbitrary expression trees to determine if they represent the same algorithm is a time consuming operation. The compute time you save avoiding any extra calls to `LambdaExpression.Compile()` are likely more than consumed by the time executing code that determines if two different expression trees result in the same executable code.

Caveats

Compiling a lambda expression to a delegate and invoking that delegate is one of the simplest operations you can perform with an expression tree. However, even with this simple operation, there are caveats you must be aware of.

Lambda Expressions create closures over any local variables that are referenced in the expression. You must guarantee that any variables that would be part of the delegate are usable at the location where you call `Compile`, and when you execute the resulting delegate. The compiler ensures that variables are in scope. However, if your expression accesses a variable that implements `IDisposable`, it's possible that your code might dispose of the object while it's still held by the expression tree.

For example, this code works fine, because `int` doesn't implement `IDisposable`:

C#

```
private static Func<int, int> CreateBoundFunc()
{
    var constant = 5; // constant is captured by the expression tree
    Expression<Func<int, int>> expression = (b) => constant + b;
    var rVal = expression.Compile();
    return rVal;
}
```

The delegate has captured a reference to the local variable `constant`. That variable is accessed at any time later, when the function returned by `CreateBoundFunc` executes.

However, consider the following (rather contrived) class that implements System.IDisposable:

C#

```
public class Resource : IDisposable
{
    private bool _isDisposed = false;
    public int Argument
    {
        get
        {
            if (!_isDisposed)
                return 5;
            else throw new ObjectDisposedException("Resource");
        }
    }

    public void Dispose()
    {
        _isDisposed = true;
    }
}
```

If you use it in an expression as shown in the following code, you get a System.ObjectDisposedException when you execute the code referenced by the `Resource.Argument` property:

C#

```
private static Func<int, int> CreateBoundResource()
{
    using (var constant = new Resource()) // constant is captured by the
expression tree
    {
        Expression<Func<int, int>> expression = (b) => constant.Argument +
b;
        var rVal = expression.Compile();
        return rVal;
    }
}
```

The delegate returned from this method has closed over the `constant` object, which has been disposed of. (It's been disposed, because it was declared in a `using` statement.)

Now, when you execute the delegate returned from this method, you have an `ObjectDisposedException` thrown at the point of execution.

It does seem strange to have a runtime error representing a compile-time construct, but that's the world you enter when you work with expression trees.

There are numerous permutations of this problem, so it's hard to offer general guidance to avoid it. Be careful about accessing local variables when defining expressions, and be careful about accessing state in the current object (represented by `this`) when creating an expression tree returned via a public API.

The code in your expression may reference methods or properties in other assemblies. That assembly must be accessible when the expression is defined, when it's compiled, and when the resulting delegate is invoked. You're met with a `ReferencedAssemblyNotFoundException` in cases where it isn't present.

Summary

Expression Trees that represent lambda expressions can be compiled to create a delegate that you can execute. Expression trees provide one mechanism to execute the code represented by an expression tree.

The Expression Tree does represent the code that would execute for any given construct you create. As long as the environment where you compile and execute the code matches the environment where you create the expression, everything works as expected. When that doesn't happen, the errors are predictable, and they're caught in your first tests of any code using the expression trees.

Interpret expressions

Article • 03/09/2023

The following code example demonstrates how the expression tree that represents the lambda expression `num => num < 5` can be decomposed into its parts.

C#

```
// 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: {param.Name} => {left.Name}
{operation.NodeType} {right.Value}");

// This code produces the following output:

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

Now, let's write some code to examine the structure of an *expression tree*. Every node in an expression tree is an object of a class that is derived from `Expression`.

That design makes visiting all the nodes in an expression tree a relatively straightforward recursive operation. The general strategy is to start at the root node and determine what kind of node it is.

If the node type has children, recursively visit the children. At each child node, repeat the process used at the root node: determine the type, and if the type has children, visit each of the children.

Examine an expression with no children

Let's start by visiting each node in a simple expression tree. Here's the code that creates a constant expression and then examines its properties:

C#

```
var constant = Expression.Constant(24, typeof(int));

Console.WriteLine($"This is a/an {constant.NodeType} expression type");
Console.WriteLine($"The type of the constant value is {constant.Type}");
Console.WriteLine($"The value of the constant value is {constant.Value}");
```

The preceding code prints the following output:

Output

```
This is a/an Constant expression type
The type of the constant value is System.Int32
The value of the constant value is 24
```

Now, let's write the code that would examine this expression and write out some important properties about it.

Addition expression

Let's start with the addition sample from the introduction to this section.

C#

```
Expression<Func<int>> sum = () => 1 + 2;
```

> **Note**
>
> Don't use `var` to declare this expression tree, because the natural type of the delegate is `Func<int>`, not `Expression<Func<int>>`.

The root node is a `LambdaExpression`. In order to get the interesting code on the right-hand side of the `=>` operator, you need to find one of the children of the `LambdaExpression`. You do that with all the expressions in this section. The parent node does help us find the return type of the `LambdaExpression`.

To examine each node in this expression, you need to recursively visit many nodes. Here's a simple first implementation:

C#

```
Expression<Func<int, int, int>> addition = (a, b) => a + b;
```

```
Console.WriteLine($"This expression is a {addition.NodeType} expression
type");
Console.WriteLine($"The name of the lambda is {((addition.Name == null) ? "
<null>" : addition.Name)}");
Console.WriteLine($"The return type is {addition.ReturnType.ToString()}");
Console.WriteLine($"The expression has {addition.Parameters.Count}
arguments. They are:");
foreach (var argumentExpression in addition.Parameters)
{
    Console.WriteLine($"\tParameter Type:
{argumentExpression.Type.ToString()}, Name: {argumentExpression.Name}");
}

var additionBody = (BinaryExpression)addition.Body;
Console.WriteLine($"The body is a {additionBody.NodeType} expression");
Console.WriteLine($"The left side is a {additionBody.Left.NodeType}
expression");
var left = (ParameterExpression)additionBody.Left;
Console.WriteLine($"\tParameter Type: {left.Type.ToString()}, Name:
{left.Name}");
Console.WriteLine($"The right side is a {additionBody.Right.NodeType}
expression");
var right = (ParameterExpression)additionBody.Right;
Console.WriteLine($"\tParameter Type: {right.Type.ToString()}, Name:
{right.Name}");
```

This sample prints the following output:

Output

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 arguments. They are:
        Parameter Type: System.Int32, Name: a
        Parameter Type: System.Int32, Name: b
The body is a/an Add expression
The left side is a Parameter expression
        Parameter Type: System.Int32, Name: a
The right side is a Parameter expression
        Parameter Type: System.Int32, Name: b
```

You notice much repetition in the preceding code sample. Let's clean that up and build a more general purpose expression node visitor. That's going to require us to write a recursive algorithm. Any node could be of a type that might have children. Any node that has children requires us to visit those children and determine what that node is. Here's the cleaned up version that utilizes recursion to visit the addition operations:

C#

```
using System.Linq.Expressions;

namespace Visitors;
// Base Visitor class:
public abstract class Visitor
{
    private readonly Expression node;

    protected Visitor(Expression node) => this.node = node;

    public abstract void Visit(string prefix);

    public ExpressionType NodeType => node.NodeType;
    public static Visitor CreateFromExpression(Expression node) =>
        node.NodeType switch
        {
            ExpressionType.Constant => new
ConstantVisitor((ConstantExpression)node),
            ExpressionType.Lambda => new
LambdaVisitor((LambdaExpression)node),
            ExpressionType.Parameter => new
ParameterVisitor((ParameterExpression)node),
            ExpressionType.Add => new BinaryVisitor((BinaryExpression)node),
            _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
        };
}

// Lambda Visitor
public class LambdaVisitor : Visitor
{
    private readonly LambdaExpression node;
    public LambdaVisitor(LambdaExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression type");
        Console.WriteLine($"{prefix}The name of the lambda is {((node.Name
== null) ? "<null>" : node.Name)}");
        Console.WriteLine($"{prefix}The return type is {node.ReturnType}");
        Console.WriteLine($"{prefix}The expression has
{node.Parameters.Count} argument(s). They are:");
        // Visit each parameter:
        foreach (var argumentExpression in node.Parameters)
        {
            var argumentVisitor = CreateFromExpression(argumentExpression);
            argumentVisitor.Visit(prefix + "\t");
        }
        Console.WriteLine($"{prefix}The expression body is:");
        // Visit the body:
        var bodyVisitor = CreateFromExpression(node.Body);
        bodyVisitor.Visit(prefix + "\t");
```

```
    }
}

// Binary Expression Visitor:
public class BinaryVisitor : Visitor
{
    private readonly BinaryExpression node;
    public BinaryVisitor(BinaryExpression node) : base(node) => this.node =
node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This binary expression is a {NodeType}
expression");
        var left = CreateFromExpression(node.Left);
        Console.WriteLine($"{prefix}The Left argument is:");
        left.Visit(prefix + "\t");
        var right = CreateFromExpression(node.Right);
        Console.WriteLine($"{prefix}The Right argument is:");
        right.Visit(prefix + "\t");
    }
}

// Parameter visitor:
public class ParameterVisitor : Visitor
{
    private readonly ParameterExpression node;
    public ParameterVisitor(ParameterExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}Type: {node.Type}, Name: {node.Name},
ByRef: {node.IsByRef}");
    }
}

// Constant visitor:
public class ConstantVisitor : Visitor
{
    private readonly ConstantExpression node;
    public ConstantVisitor(ConstantExpression node) : base(node) =>
this.node = node;

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This is an {NodeType} expression type");
        Console.WriteLine($"{prefix}The type of the constant value is
{node.Type}");
        Console.WriteLine($"{prefix}The value of the constant value is
{node.Value}");
```

```
    }
}
```

This algorithm is the basis of an algorithm that visits any arbitrary `LambdaExpression`. The code you created only looks for a small sample of the possible sets of expression tree nodes that it may encounter. However, you can still learn quite a bit from what it produces. (The default case in the `Visitor.CreateFromExpression` method prints a message to the error console when a new node type is encountered. That way, you know to add a new expression type.)

When you run this visitor on the preceding addition expression, you get the following output:

Output

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This is an Parameter expression type
                Type: System.Int32, Name: b, ByRef: False
```

Now that you've built a more general visitor implementation, you can visit and process many more different types of expressions.

Addition Expression with more operands

Let's try a more complicated example, yet still limit the node types to addition only:

C#

```
Expression<Func<int>> sum = () => 1 + 2 + 3 + 4;
```

Before you run these examples on the visitor algorithm, try a thought exercise to work out what the output might be. Remember that the `+` operator is a *binary operator*: it

must have two children, representing the left and right operands. There are several possible ways to construct a tree that could be correct:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
Expression<Func<int>> sum2 = () => ((1 + 2) + 3) + 4;

Expression<Func<int>> sum3 = () => (1 + 2) + (3 + 4);
Expression<Func<int>> sum4 = () => 1 + ((2 + 3) + 4);
Expression<Func<int>> sum5 = () => (1 + (2 + 3)) + 4;
```

You can see the separation into two possible answers to highlight the most promising. The first represents *right associative* expressions. The second represent *left associative* expressions. The advantage of both of those two formats is that the format scales to any arbitrary number of addition expressions.

If you do run this expression through the visitor, you see this output, verifying that the simple addition expression is *left associative*.

In order to run this sample, and see the full expression tree, you make one change to the source expression tree. When the expression tree contains all constants, the resulting tree simply contains the constant value of `10`. The compiler performs all the addition and reduces the expression to its simplest form. Simply adding one variable in the expression is sufficient to see the original tree:

C#

```
Expression<Func<int, int>> sum = (a) => 1 + a + 3 + 4;
```

Create a visitor for this sum and run the visitor you see this output:

Output

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This binary expression is a Add expression
                        The Left argument is:
```

```
                                This is an Constant expression type
                                The type of the constant value is
System.Int32
                                The value of the constant value is 1
                        The Right argument is:
                                This is an Parameter expression type
                                Type: System.Int32, Name: a, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
        The Right argument is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 4
```

You can run any of the other samples through the visitor code and see what tree it represents. Here's an example of the preceding `sum3` expression (with an additional parameter to prevent the compiler from computing the constant):

C#

```
Expression<Func<int, int, int>> sum3 = (a, b) => (1 + a) + (3 + b);
```

Here's the output from the visitor:

Output

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 2 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: a, ByRef: False
        This is an Parameter expression type
        Type: System.Int32, Name: b, ByRef: False
The expression body is:
        This binary expression is a Add expression
        The Left argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 1
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: a, ByRef: False
        The Right argument is:
                This binary expression is a Add expression
                The Left argument is:
                        This is an Constant expression type
```

```
                        The type of the constant value is System.Int32
                        The value of the constant value is 3
                The Right argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: b, ByRef: False
```

Notice that the parentheses aren't part of the output. There are no nodes in the expression tree that represent the parentheses in the input expression. The structure of the expression tree contains all the information necessary to communicate the precedence.

Extending this sample

The sample deals with only the most rudimentary expression trees. The code you've seen in this section only handles constant integers and the binary `+` operator. As a final sample, let's update the visitor to handle a more complicated expression. Let's make it work for the following factorial expression:

C#

```
Expression<Func<int, int>> factorial = (n) =>
    n == 0 ?
    1 :
    Enumerable.Range(1, n).Aggregate((product, factor) => product * factor);
```

This code represents one possible implementation for the mathematical *factorial* function. The way you've written this code highlights two limitations of building expression trees by assigning lambda expressions to Expressions. First, statement lambdas aren't allowed. That means you can't use loops, blocks, if / else statements, and other control structures common in C#. You're limited to using expressions. Second, you can't recursively call the same expression. You could if it were already a delegate, but you can't call it in its expression tree form. In the section on building expression trees, you learn techniques to overcome these limitations.

In this expression, you encounter nodes of all these types:

1. Equal (binary expression)
2. Multiply (binary expression)
3. Conditional (the `? :` expression)
4. Method Call Expression (calling `Range()` and `Aggregate()`)

One way to modify the visitor algorithm is to keep executing it, and write the node type every time you reach your `default` clause. After a few iterations, you've see each of the

potential nodes. Then, you have all you need. The result would be something like this:

C#

```
public static Visitor CreateFromExpression(Expression node) =>
    node.NodeType switch
    {
        ExpressionType.Constant    => new
ConstantVisitor((ConstantExpression)node),
        ExpressionType.Lambda      => new
LambdaVisitor((LambdaExpression)node),
        ExpressionType.Parameter   => new
ParameterVisitor((ParameterExpression)node),
        ExpressionType.Add         => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Equal       => new
BinaryVisitor((BinaryExpression)node),
        ExpressionType.Multiply    => new BinaryVisitor((BinaryExpression)
node),
        ExpressionType.Conditional => new
ConditionalVisitor((ConditionalExpression) node),
        ExpressionType.Call        => new
MethodCallVisitor((MethodCallExpression) node),
        _ => throw new NotImplementedException($"Node not processed yet:
{node.NodeType}"),
    };
```

The `ConditionalVisitor` and `MethodCallVisitor` process those two nodes:

C#

```
public class ConditionalVisitor : Visitor
{
    private readonly ConditionalExpression node;
    public ConditionalVisitor(ConditionalExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        var testVisitor = Visitor.CreateFromExpression(node.Test);
        Console.WriteLine($"{prefix}The Test for this expression is:");
        testVisitor.Visit(prefix + "\t");
        var trueVisitor = Visitor.CreateFromExpression(node.IfTrue);
        Console.WriteLine($"{prefix}The True clause for this expression
is:");
        trueVisitor.Visit(prefix + "\t");
        var falseVisitor = Visitor.CreateFromExpression(node.IfFalse);
        Console.WriteLine($"{prefix}The False clause for this expression
```

```
is:");
        falseVisitor.Visit(prefix + "\t");
    }
}

public class MethodCallVisitor : Visitor
{
    private readonly MethodCallExpression node;
    public MethodCallVisitor(MethodCallExpression node) : base(node)
    {
        this.node = node;
    }

    public override void Visit(string prefix)
    {
        Console.WriteLine($"{prefix}This expression is a {NodeType}
expression");
        if (node.Object == null)
            Console.WriteLine($"{prefix}This is a static method call");
        else
        {
            Console.WriteLine($"{prefix}The receiver (this) is:");
            var receiverVisitor = Visitor.CreateFromExpression(node.Object);
            receiverVisitor.Visit(prefix + "\t");
        }

        var methodInfo = node.Method;
        Console.WriteLine($"{prefix}The method name is
{methodInfo.DeclaringType}.{methodInfo.Name}");
        // There is more here, like generic arguments, and so on.
        Console.WriteLine($"{prefix}The Arguments are:");
        foreach (var arg in node.Arguments)
        {
            var argVisitor = Visitor.CreateFromExpression(arg);
            argVisitor.Visit(prefix + "\t");
        }
    }
}
```

And the output for the expression tree would be:

Output

```
This expression is a/an Lambda expression type
The name of the lambda is <null>
The return type is System.Int32
The expression has 1 argument(s). They are:
        This is an Parameter expression type
        Type: System.Int32, Name: n, ByRef: False
The expression body is:
        This expression is a Conditional expression
        The Test for this expression is:
                This binary expression is a Equal expression
```

```
                The Left argument is:
                        This is an Parameter expression type
                        Type: System.Int32, Name: n, ByRef: False
                The Right argument is:
                        This is an Constant expression type
                        The type of the constant value is System.Int32
                        The value of the constant value is 0
        The True clause for this expression is:
                This is an Constant expression type
                The type of the constant value is System.Int32
                The value of the constant value is 1
        The False clause for this expression is:
                This expression is a Call expression
                This is a static method call
                The method name is System.Linq.Enumerable.Aggregate
                The Arguments are:
                        This expression is a Call expression
                        This is a static method call
                        The method name is System.Linq.Enumerable.Range
                        The Arguments are:
                                This is an Constant expression type
                                The type of the constant value is
System.Int32
                                The value of the constant value is 1
                                This is an Parameter expression type
                                Type: System.Int32, Name: n, ByRef: False
                        This expression is a Lambda expression type
                        The name of the lambda is <null>
                        The return type is System.Int32
                        The expression has 2 arguments. They are:
                                This is an Parameter expression type
                                Type: System.Int32, Name: product, ByRef:
False
                                This is an Parameter expression type
                                Type: System.Int32, Name: factor, ByRef:
False
                        The expression body is:
                                This binary expression is a Multiply
expression
                                The Left argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: product,
ByRef: False
                                The Right argument is:
                                        This is an Parameter expression type
                                        Type: System.Int32, Name: factor,
ByRef: False
```

Extend the Sample Library

The samples in this section show the core techniques to visit and examine nodes in an expression tree. It simplified the types of nodes you'll encounter to concentrate on the

core tasks of visiting and accessing nodes in an expression tree.

First, the visitors only handle constants that are integers. Constant values could be any other numeric type, and the C# language supports conversions and promotions between those types. A more robust version of this code would mirror all those capabilities.

Even the last example recognizes a subset of the possible node types. You can still feed it many expressions that cause it to fail. A full implementation is included in .NET Standard under the name ExpressionVisitor and can handle all the possible node types.

Finally, the library used in this article was built for demonstration and learning. It's not optimized. It makes the structures clear, and to highlight the techniques used to visit the nodes and analyze what's there.

Even with those limitations, you should be well on your way to writing algorithms that read and understand expression trees.

Build expression trees

Article • 03/09/2023

The C# compiler created all the expression trees you've seen so far. You created a lambda expression assigned to a variable typed as an `Expression<Func<T>>` or some similar type. For many scenarios, you build an expression in memory at run time.

Expression trees are immutable. Being immutable means that you must build the tree from the leaves up to the root. The APIs you use to build expression trees reflect this fact: The methods you use to build a node take all its children as arguments. Let's walk through a few examples to show you the techniques.

Create nodes

You start with the addition expression you've been working with throughout these sections:

```
C#

Expression<Func<int>> sum = () => 1 + 2;
```

To construct that expression tree, you first construct the leaf nodes. The leaf nodes are constants. Use the Constant method to create the nodes:

```
C#

var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
```

Next, build the addition expression:

```
C#

var addition = Expression.Add(one, two);
```

Once you've built the addition expression, you create the lambda expression:

```
C#

var lambda = Expression.Lambda(addition);
```

This lambda expression contains no arguments. Later in this section, you see how to map arguments to parameters and build more complicated expressions.

For expressions like this one, you may combine all the calls into a single statement:

```
C#
```

```
var lambda2 = Expression.Lambda(
    Expression.Add(
        Expression.Constant(1, typeof(int)),
        Expression.Constant(2, typeof(int))
    )
);
```

Build a tree

The previous section showed the basics of building an expression tree in memory. More complex trees generally mean more node types, and more nodes in the tree. Let's run through one more example and show two more node types that you typically build when you create expression trees: the argument nodes, and method call nodes. Let's build an expression tree to create this expression:

```
C#
```

```
Expression<Func<double, double, double>> distanceCalc =
    (x, y) => Math.Sqrt(x * x + y * y);
```

You start by creating parameter expressions for `x` and `y`:

```
C#
```

```
var xParameter = Expression.Parameter(typeof(double), "x");
var yParameter = Expression.Parameter(typeof(double), "y");
```

Creating the multiplication and addition expressions follows the pattern you've already seen:

```
C#
```

```
var xSquared = Expression.Multiply(xParameter, xParameter);
var ySquared = Expression.Multiply(yParameter, yParameter);
var sum = Expression.Add(xSquared, ySquared);
```

Next, you need to create a method call expression for the call to `Math.Sqrt`.

C#

```
var sqrtMethod = typeof(Math).GetMethod("Sqrt", new[] { typeof(double) }) ??
throw new InvalidOperationException("Math.Sqrt not found!");
var distance = Expression.Call(sqrtMethod, sum);
```

The `GetMethod` call could return `null` if the method isn't found. Most likely that's because you've misspelled the method name. Otherwise, it could mean the required assembly isn't loaded. Finally, you put the method call into a lambda expression, and make sure to define the arguments to the lambda expression:

C#

```
var distanceLambda = Expression.Lambda(
    distance,
    xParameter,
    yParameter);
```

In this more complicated example, you see a couple more techniques that you often need to create expression trees.

First, you need to create the objects that represent parameters or local variables before you use them. Once you've created those objects, you can use them in your expression tree wherever you need.

Second, you need to use a subset of the Reflection APIs to create a System.Reflection.MethodInfo object so that you can create an expression tree to access that method. You must limit yourself to the subset of the Reflection APIs that are available on the .NET Core platform. Again, these techniques extend to other expression trees.

Build code in depth

You aren't limited in what you can build using these APIs. However, the more complicated expression tree that you want to build, the more difficult the code is to manage and to read.

Let's build an expression tree that is the equivalent of this code:

C#

```
Func<int, int> factorialFunc = (n) =>
{
    var res = 1;
```

```
    while (n > 1)
    {
        res = res * n;
        n--;
    }
    return res;
};
```

The preceding code didn't build the expression tree, but simply the delegate. Using the `Expression` class, you can't build statement lambdas. Here's the code that is required to build the same functionality. There isn't an API for building a `while` loop, instead you need to build a loop that contains a conditional test, and a label target to break out of the loop.

C#

```
var nArgument = Expression.Parameter(typeof(int), "n");
var result = Expression.Variable(typeof(int), "result");

// Creating a label that represents the return value
LabelTarget label = Expression.Label(typeof(int));

var initializeResult = Expression.Assign(result, Expression.Constant(1));

// This is the inner block that performs the multiplication,
// and decrements the value of 'n'
var block = Expression.Block(
    Expression.Assign(result,
        Expression.Multiply(result, nArgument)),
    Expression.PostDecrementAssign(nArgument)
);

// Creating a method body.
BlockExpression body = Expression.Block(
    new[] { result },
    initializeResult,
    Expression.Loop(
        Expression.IfThenElse(
            Expression.GreaterThan(nArgument, Expression.Constant(1)),
            block,
            Expression.Break(label, result)
        ),
        label
    )
);
```

The code to build the expression tree for the factorial function is quite a bit longer, more complicated, and it's riddled with labels and break statements and other elements you'd like to avoid in our everyday coding tasks.

For this section, you wrote code to visit every node in this expression tree and write out information about the nodes that are created in this sample. You can view or download the sample code at the dotnet/docs GitHub repository. Experiment for yourself by building and running the samples.

Map code constructs to expressions

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

C#

```
// 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 });
```

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.

C#

```
// 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.

Translate expression trees

Article • 03/09/2023

In this article, you learn how to visit each node in an expression tree while building a modified copy of that expression tree. You translate expression trees to understand the algorithms so that it can be translated into another environment. You change the algorithm that has been created. You might add logging, intercept method calls and track them, or other purposes.

The code you build to translate an expression tree is an extension of what you've already seen to visit all the nodes in a tree. When you translate an expression tree, you visit all the nodes, and while visiting them, build the new tree. The new tree may contain references to the original nodes, or new nodes that you've placed in the tree.

Let's visit an expression tree, and creating a new tree with some replacement nodes. In this example, let's replace any constant with a constant that is 10 times larger. Otherwise, you leave the expression tree intact. Rather than reading the value of the constant and replacing it with a new constant, you make this replacement by replacing the constant node with a new node that performs the multiplication.

Here, once you find a constant node, you create a new multiplication node whose children are the original constant and the constant `10`:

C#

```
private static Expression ReplaceNodes(Expression original)
{
    if (original.NodeType == ExpressionType.Constant)
    {
        return Expression.Multiply(original, Expression.Constant(10));
    }
    else if (original.NodeType == ExpressionType.Add)
    {
        var binaryExpression = (BinaryExpression)original;
        return Expression.Add(
            ReplaceNodes(binaryExpression.Left),
            ReplaceNodes(binaryExpression.Right));
    }
    return original;
}
```

Create a new tree by replacing the original node with the substitute. You verify the changes by compiling and executing the replaced tree.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var addition = Expression.Add(one, two);
var sum = ReplaceNodes(addition);
var executableFunc = Expression.Lambda(sum);

var func = (Func<int>)executableFunc.Compile();
var answer = func();
Console.WriteLine(answer);
```

Building a new tree is a combination of visiting the nodes in the existing tree, and creating new nodes and inserting them into the tree. The previous example shows the importance of expression trees being immutable. Notice that the new tree created in the preceding code contains a mixture of newly created nodes, and nodes from the existing tree. Nodes can be used in both trees because the nodes in the existing tree can't be modified. Reusing nodes results in significant memory efficiencies. The same nodes can be used throughout a tree, or in multiple expression trees. Since nodes can't be modified, the same node can be reused whenever it's needed.

Traverse and execute an addition

Let's verify the new tree by building a second visitor that walks the tree of addition nodes and computes the result. Make a couple modifications to the visitor that you've seen so far. In this new version, the visitor returns the partial sum of the addition operation up to this point. For a constant expression it's simply the value of the constant expression. For an addition expression, the result is the sum of the left and right operands, once those trees have been traversed.

C#

```
var one = Expression.Constant(1, typeof(int));
var two = Expression.Constant(2, typeof(int));
var three = Expression.Constant(3, typeof(int));
var four = Expression.Constant(4, typeof(int));
var addition = Expression.Add(one, two);
var add2 = Expression.Add(three, four);
var sum = Expression.Add(addition, add2);

// Declare the delegate, so you can call it
// from itself recursively:
Func<Expression, int> aggregate = null!;
// Aggregate, return constants, or the sum of the left and right operand.
// Major simplification: Assume every binary expression is an addition.
aggregate = (exp) =>
    exp.NodeType == ExpressionType.Constant ?
    (int)((ConstantExpression)exp).Value :
```

```
    aggregate(((BinaryExpression)exp).Left) +
aggregate(((BinaryExpression)exp).Right);

var theSum = aggregate(sum);
Console.WriteLine(theSum);
```

There's quite a bit of code here, but the concepts are approachable. This code visits children in a depth first search. When it encounters a constant node, the visitor returns the value of the constant. After the visitor has visited both children, those children have computed the sum computed for that subtree. The addition node can now compute its sum. Once all the nodes in the expression tree have been visited, the sum has been computed. You can trace the execution by running the sample in the debugger and tracing the execution.

Let's make it easier to trace how the nodes are analyzed and how the sum is computed by traversing the tree. Here's an updated version of the Aggregate method that includes quite a bit of tracing information:

C#

```
private static int Aggregate(Expression exp)
{
    if (exp.NodeType == ExpressionType.Constant)
    {
        var constantExp = (ConstantExpression)exp;
        Console.Error.WriteLine($"Found Constant: {constantExp.Value}");
        if (constantExp.Value is int value)
        {
            return value;
        }
        else
        {
            return 0;
        }
    }
    else if (exp.NodeType == ExpressionType.Add)
    {
        var addExp = (BinaryExpression)exp;
        Console.Error.WriteLine("Found Addition Expression");
        Console.Error.WriteLine("Computing Left node");
        var leftOperand = Aggregate(addExp.Left);
        Console.Error.WriteLine($"Left is: {leftOperand}");
        Console.Error.WriteLine("Computing Right node");
        var rightOperand = Aggregate(addExp.Right);
        Console.Error.WriteLine($"Right is: {rightOperand}");
        var sum = leftOperand + rightOperand;
        Console.Error.WriteLine($"Computed sum: {sum}");
        return sum;
    }
```

```
    else throw new NotSupportedException("Haven't written this yet");
}
```

Running it on the `sum` expression yields the following output:

Output

```
10
Found Addition Expression
Computing Left node
Found Addition Expression
Computing Left node
Found Constant: 1
Left is: 1
Computing Right node
Found Constant: 2
Right is: 2
Computed sum: 3
Left is: 3
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 10
10
```

Trace the output and follow along in the preceding code. You should be able to work out how the code visits each node and computes the sum as it goes through the tree and finds the sum.

Now, let's look at a different run, with the expression given by `sum1`:

C#

```
Expression<Func<int>> sum1 = () => 1 + (2 + (3 + 4));
```

Here's the output from examining this expression:

Output

```
Found Addition Expression
Computing Left node
Found Constant: 1
```

```
Left is: 1
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 2
Left is: 2
Computing Right node
Found Addition Expression
Computing Left node
Found Constant: 3
Left is: 3
Computing Right node
Found Constant: 4
Right is: 4
Computed sum: 7
Right is: 7
Computed sum: 9
Right is: 9
Computed sum: 10
10
```

While the final answer is the same, the tree traversal is different. The nodes are traveled in a different order, because the tree was constructed with different operations occurring first.

Create a modified copy

Create a new **Console Application** project. Add a `using` directive to the file for the `System.Linq.Expressions` namespace. Add the `AndAlsoModifier` class to your project.

C#

```
public class AndAlsoModifier : ExpressionVisitor
{
    public Expression Modify(Expression expression)
    {
        return Visit(expression);
    }

    protected override Expression VisitBinary(BinaryExpression b)
    {
        if (b.NodeType == ExpressionType.AndAlso)
        {
            Expression left = this.Visit(b.Left);
            Expression right = this.Visit(b.Right);

            // Make this binary expression an OrElse operation instead of an
AndAlso operation.
            return Expression.MakeBinary(ExpressionType.OrElse, left, right,
b.IsLiftedToNull, b.Method);
```

```
        }

        return base.VisitBinary(b);
    }
}
```

This class inherits the ExpressionVisitor class and is specialized to modify expressions that represent conditional `AND` operations. It changes these operations from a conditional `AND` to a conditional `OR`. The class overrides the VisitBinary method of the base type, because conditional `AND` expressions are represented as binary expressions. In the `VisitBinary` method, if the expression that is passed to it represents a conditional `AND` operation, the code constructs a new expression that contains the conditional `OR` operator instead of the conditional `AND` operator. If the expression that is passed to `VisitBinary` doesn't represent a conditional `AND` operation, the method defers to the base class implementation. The base class methods construct nodes that are like the expression trees that are passed in, but the nodes have their sub trees replaced with the expression trees produced recursively by the visitor.

Add a `using` directive to the file for the `System.Linq.Expressions` namespace. Add code to the `Main` method in the Program.cs file to create an expression tree and pass it to the method that modifies it.

C#

```
Expression<Func<string, bool>> expr = name => name.Length > 10 &&
name.StartsWith("G");
Console.WriteLine(expr);

AndAlsoModifier treeModifier = new AndAlsoModifier();
Expression modifiedExpr = treeModifier.Modify((Expression)expr);

Console.WriteLine(modifiedExpr);

/*  This code produces the following output:

    name => ((name.Length > 10) && name.StartsWith("G"))
    name => ((name.Length > 10) || name.StartsWith("G"))
*/
```

The code creates an expression that contains a conditional `AND` operation. It then creates an instance of the `AndAlsoModifier` class and passes the expression to the `Modify` method of this class. Both the original and the modified expression trees are outputted to show the change. Compile and run the application.

Learn more

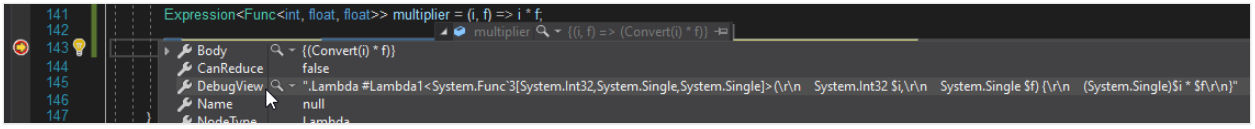
This sample shows a small subset of the code you would build to traverse and interpret the algorithms represented by an expression tree. For information on building a general purpose library that translates expression trees into another language, read this series by Matt Warren. It goes into great detail on how to translate any of the code you might find in an expression tree.

You've now seen the true power of expression trees. You examine a set of code, make any changes you'd like to that code, and execute the changed version. Because the expression trees are immutable, you create new trees by using the components of existing trees. Reusing nodes minimizes the amount of memory needed to create modified expression trees.

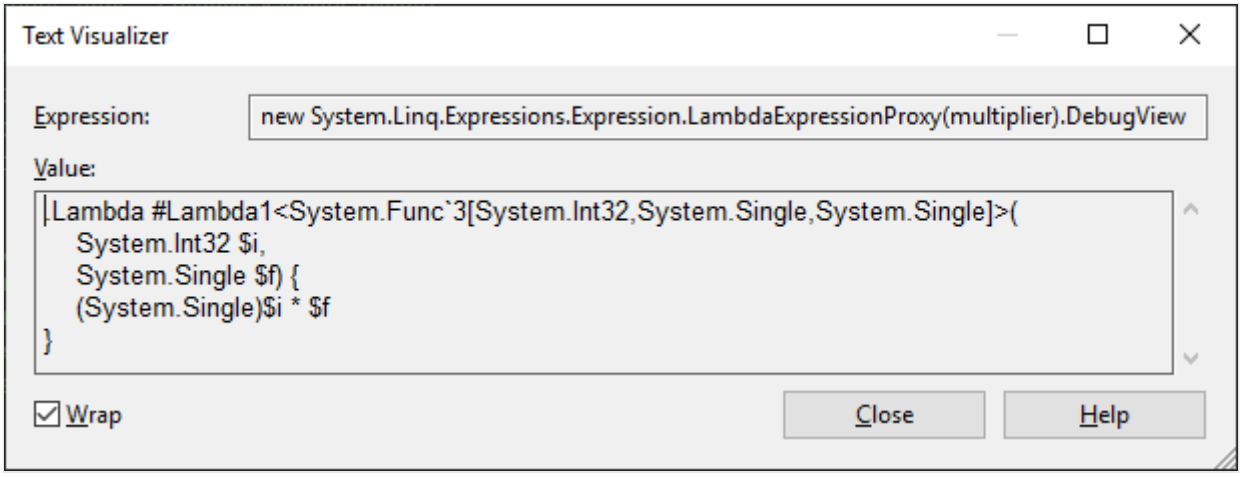
Debugging expression trees in Visual Studio

Article • 03/09/2023

You can analyze the structure and content of expression trees when you debug your applications. To get a quick overview of the expression tree structure, you can use the `DebugView` property, which represents expression trees using a special syntax. `DebugView` is available only in debug mode.

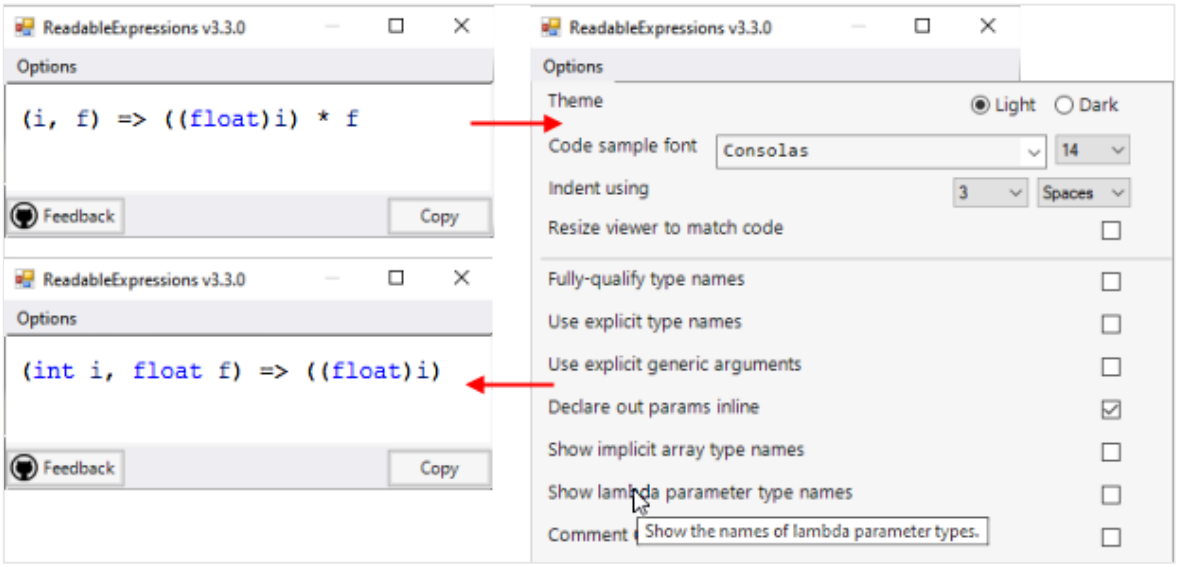


Since `DebugView` is a string, you can use the built-in Text Visualizer to view it across multiple lines, by selecting **Text Visualizer** from the magnifying glass icon next to the `DebugView` label.

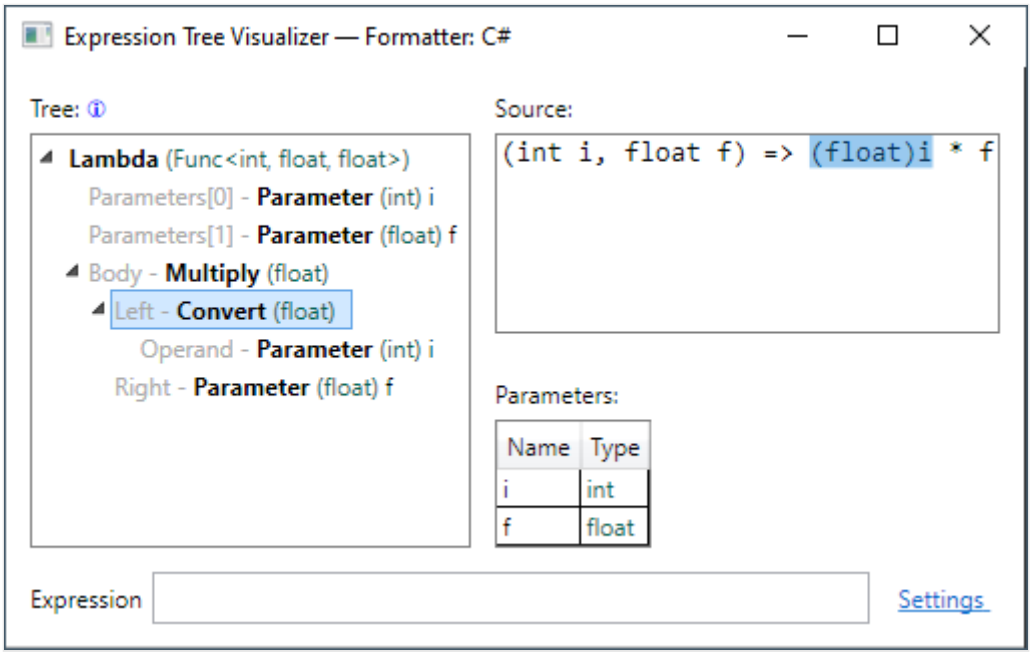


Alternatively, you can install and use a custom visualizer for expression trees, such as:

- Readable Expressions (MIT license, available at the Visual Studio Marketplace), renders the expression tree as themeable C# code, with various rendering options:

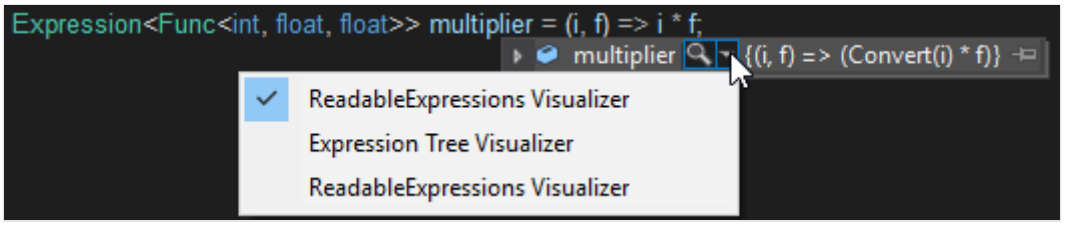


- Expression Tree Visualizer (MIT license) provides a tree view of the expression tree and its individual nodes:



Open a visualizer for an expression tree

Select the magnifying glass icon that appears next to the expression tree in **DataTips**, a **Watch** window, the **Autos** window, or the **Locals** window. A list of available visualizers is displayed:



Select the visualizer you want to use.

See also

- Debugging in Visual Studio
- Create Custom Visualizers
- DebugView syntax

DebugView syntax

Article • 03/09/2023

The **DebugView** property (available only when debugging) provides a string rendering of expression trees. Most of the syntax is fairly straightforward to understand; the special cases are described in the following sections.

Each example is followed by a block comment, containing the **DebugView**.

ParameterExpression

ParameterExpression variable names are displayed with a `$` symbol at the beginning.

If a parameter doesn't have a name, it's assigned an automatically generated name, such as `$var1` or `$var2`.

```
C#

ParameterExpression numParam =  Expression.Parameter(typeof(int), "num");
/*
    $num
*/

ParameterExpression numParam =  Expression.Parameter(typeof(int));
/*
    $var1
*/
```

ConstantExpression

For ConstantExpression objects that represent integer values, strings, and `null`, the value of the constant is displayed.

For numeric types that have standard suffixes as C# literals, the suffix is added to the value. The following table shows the suffixes associated with various numeric types.

Expand table

Type	Keyword	Suffix
System.UInt32	uint	U
System.Int64	long	L

Type	Keyword	Suffix
System.UInt64	ulong	UL
System.Double	double	D
System.Single	float	F
System.Decimal	decimal	M

C#

```
int num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10
*/

double num = 10;
ConstantExpression expr = Expression.Constant(num);
/*
    10D
*/
```

BlockExpression

If the type of a BlockExpression object differs from the type of the last expression in the block, the type is displayed within angle brackets (`<` and `>`). Otherwise, the type of the BlockExpression object isn't displayed.

C#

```
BlockExpression block = Expression.Block(Expression.Constant("test"));
/*
    .Block() {
        "test"
    }
*/

BlockExpression block =  Expression.Block(typeof(Object),
Expression.Constant("test"));
/*
    .Block<System.Object>() {
        "test"
    }
*/
```

LambdaExpression

LambdaExpression objects are displayed together with their delegate types.

If a lambda expression doesn't have a name, it's assigned an automatically generated name, such as `#Lambda1` or `#Lambda2`.

C#

```
LambdaExpression lambda =  Expression.Lambda<Func<int>>
(Expression.Constant(1));
/*
    .Lambda #Lambda1<System.Func'1[System.Int32]>() {
        1
    }
*/

LambdaExpression lambda =  Expression.Lambda<Func<int>>
(Expression.Constant(1), "SampleLambda", null);
/*
    .Lambda #SampleLambda<System.Func'1[System.Int32]>() {
        1
    }
*/
```

LabelExpression

If you specify a default value for the LabelExpression object, this value is displayed before the LabelTarget object.

The `.Label` token indicates the start of the label. The `.LabelTarget` token indicates the destination of the target to jump to.

If a label doesn't have a name, it's assigned an automatically generated name, such as `#Label1` or `#Label2`.

C#

```
LabelTarget target = Expression.Label(typeof(int), "SampleLabel");
BlockExpression block = Expression.Block(
    Expression.Goto(target, Expression.Constant(0)),
    Expression.Label(target, Expression.Constant(-1))
);
/*
    .Block() {
        .Goto SampleLabel { 0 };
        .Label
            -1
```

```
        .LabelTarget SampleLabel:
    }
*/

LabelTarget target = Expression.Label();
BlockExpression block = Expression.Block(
    Expression.Goto(target),
    Expression.Label(target)
);
/*
    .Block() {
        .Goto #Label1 { };
        .Label
        .LabelTarget #Label1:
    }
*/
```

Checked Operators

Checked operators are displayed with the `#` symbol in front of the operator. For example, the checked addition operator is displayed as `#+`.

C#

```
Expression expr = Expression.AddChecked( Expression.Constant(1),
Expression.Constant(2));
/*
    1 #+ 2
*/

Expression expr = Expression.ConvertChecked( Expression.Constant(10.0),
typeof(int));
/*
    #(System.Int32)10D
*/
```

System.Linq.Expressions.Expression.Add methods

Article • 01/30/2024

This article provides supplementary remarks to the reference documentation for this API.

The Add method returns a BinaryExpression that has the Method property set to the implementing method. The Type property is set to the type of the node. If the node is lifted, the IsLifted and IsLiftedToNull properties are both `true`. Otherwise, they are `false`. The Conversion property is `null`.

The following information describes the implementing method, the node type, and whether a node is lifted.

Implementing method

The following rules determine the selected implementing method for the operation:

- If the Type property of either `left` or `right` represents a user-defined type that overloads the addition operator, the MethodInfo that represents that method is the implementing method.
- Otherwise, if `left`.Type and `right`.Type are numeric types, the implementing method is `null`.

Node type and lifted versus non-lifted

If the implementing method is not `null`:

- If `left`.Type and `right`.Type are assignable to the corresponding argument types of the implementing method, the node is not lifted. The type of the node is the return type of the implementing method.

- If the following two conditions are satisfied, the node is lifted and the type of the node is the nullable type that corresponds to the return type of the implementing method:
  - `left`.Type and `right`.Type are both value types of which at least one is nullable and the corresponding non-nullable types are equal to the corresponding argument types of the implementing method.

    - The return type of the implementing method is a non-nullable value type.

If the implementing method is `null`:

- If `left`.Type and `right`.Type are both non-nullable, the node is not lifted. The type of the node is the result type of the predefined addition operator.
- If `left`.Type and `right`.Type are both nullable, the node is lifted. The type of the node is the nullable type that corresponds to the result type of the predefined addition operator.

System.Linq.Expressions.BinaryExpression class

Article • 01/09/2024

This article provides supplementary remarks to the reference documentation for this API.

The BinaryExpression class represents an expression that has a binary operator.

The following tables summarize the factory methods that can be used to create a BinaryExpression that has a specific node type, represented by the NodeType property. Each table contains information for a specific class of operations such as arithmetic or bitwise.

Binary arithmetic operations

Expand table

Node Type	Factory Method
Add	Add
AddChecked	AddChecked
Divide	Divide
Modulo	Modulo
Multiply	Multiply
MultiplyChecked	MultiplyChecked
Power	Power
Subtract	Subtract
SubtractChecked	SubtractChecked

Bitwise operations

Expand table

Node Type	Factory Method
And	And
Or	Or
ExclusiveOr	ExclusiveOr

Shift operations

Expand table

Node Type	Factory Method
LeftShift	LeftShift
RightShift	RightShift

Conditional Boolean operations

Expand table

Node Type	Factory Method
AndAlso	AndAlso
OrElse	OrElse

Comparison operations

Expand table

Node Type	Factory Method
Equal	Equal
NotEqual	NotEqual
GreaterThanOrEqual	GreaterThanOrEqual
GreaterThan	GreaterThan
LessThan	LessThan
LessThanOrEqual	LessThanOrEqual

Coalescing operations

Expand table

Node Type	Factory Method
Coalesce	Coalesce

Array indexing operations

Expand table

Node Type	Factory Method
ArrayIndex	ArrayIndex

In addition, the MakeBinary methods can also be used to create a BinaryExpression. These factory methods can be used to create a BinaryExpression of any node type that represents a binary operation. The parameter of these methods that is of type NodeType specifies the desired node type.

Interoperability Overview

Article • 02/25/2023

Interoperability enables you to preserve and take advantage of existing investments in unmanaged code. Code that runs under the control of the common language runtime (CLR) is *managed code*, and code that runs outside the CLR is *unmanaged code*. COM, COM+, C++ components, ActiveX components, and Microsoft Windows API are examples of unmanaged code.

.NET enables interoperability with unmanaged code through platform invoke services, the System.Runtime.InteropServices namespace, C++ interoperability, and COM interoperability (COM interop).

Platform Invoke

Platform invoke is a service that enables managed code to call unmanaged functions implemented in dynamic link libraries (DLLs), such as the Microsoft Windows API. It locates and invokes an exported function and marshals its arguments (integers, strings, arrays, structures, and so on) across the interoperation boundary as needed.

For more information, see Consuming Unmanaged DLL Functions and How to use platform invoke to play a WAV file.

> **Note**
>
> The Common Language Runtime (CLR) manages access to system resources. Calling unmanaged code that is outside the CLR bypasses this security mechanism, and therefore presents a security risk. For example, unmanaged code might call resources in unmanaged code directly, bypassing CLR security mechanisms. For more information, see Security in .NET.

C++ Interop

You can use C++ interop, also known as It Just Works (IJW), to wrap a native C++ class. C++ interop enables code authored in C# or another .NET language to access it. You write C++ code to wrap a native DLL or COM component. Unlike other .NET languages, Visual C++ has interoperability support that enables managed and unmanaged code in the same application and even in the same file. You then build the C++ code by using the **/clr** compiler switch to produce a managed assembly. Finally, you add a reference to

the assembly in your C# project and use the wrapped objects just as you would use other managed classes.

Exposing COM Components to C#

You can consume a COM component from a C# project. The general steps are as follows:

1. Locate a COM component to use and register it. Use regsvr32.exe to register or un–register a COM DLL.
2. Add to the project a reference to the COM component or type library. When you add the reference, Visual Studio uses the Tlbimp.exe (Type Library Importer), which takes a type library as input, to output a .NET interop assembly. The assembly, also named a runtime callable wrapper (RCW), contains managed classes and interfaces that wrap the COM classes and interfaces that are in the type library. Visual Studio adds to the project a reference to the generated assembly.
3. Create an instance of a class defined in the RCW. Creating an instance of that class creates an instance of the COM object.
4. Use the object just as you use other managed objects. When the object is reclaimed by garbage collection, the instance of the COM object is also released from memory.

For more information, see Exposing COM Components to the .NET Framework.

Exposing C# to COM

COM clients can consume C# types that have been correctly exposed. The basic steps to expose C# types are as follows:

1. Add interop attributes in the C# project. You can make an assembly COM visible by modifying C# project properties. For more information, see Assembly Information Dialog Box.
2. Generate a COM type library and register it for COM usage. You can modify C# project properties to automatically register the C# assembly for COM interop. Visual Studio uses the Regasm.exe (Assembly Registration Tool), using the `/tlb` command-line switch, which takes a managed assembly as input, to generate a type library. This type library describes the `public` types in the assembly and adds registry entries so that COM clients can create managed classes.

For more information, see Exposing .NET Framework Components to COM and Example COM Class.

See also

- Improving Interop Performance
- Introduction to Interoperability between COM and .NET
- Introduction to COM Interop in Visual Basic
- Marshaling between Managed and Unmanaged Code
- Interoperating with Unmanaged Code
- Registering Assemblies with COM

Example COM Class

Article • 02/25/2023

The following code is an example of a class that you would expose as a COM object. After you place this code in a .cs file added to your project, set the **Register for COM Interop** property to **True**. For more information, see How to: Register a Component for COM Interop.

Exposing C# objects to COM requires declaring a class interface, an "events interface" if necessary, and the class itself. Class members must follow these rules to be visible to COM:

- The class must be public.
- Properties, methods, and events must be public.
- Properties and methods must be declared on the class interface.
- Events must be declared in the event interface.

Other public members in the class that you don't declare in these interfaces aren't visible to COM, but they're visible to other .NET objects. To expose properties and methods to COM, you must declare them on the class interface and mark them with a `DispId` attribute, and implement them in the class. The order in which you declare the members in the interface is the order used for the COM vtable. To expose events from your class, you must declare them on the events interface and mark them with a `DispId` attribute. The class shouldn't implement this interface.

The class implements the class interface; it can implement more than one interface, but the first implementation is the default class interface. Implement the methods and properties exposed to COM here. They must be public and must match the declarations in the class interface. Also, declare the events raised by the class here. They must be public and must match the declarations in the events interface.

Example

C#

```
using System.Runtime.InteropServices;

namespace project_name
{
    [Guid("EAA4976A-45C3-4BC5-BC0B-E474F4C3C83F")]
    public interface ComClass1Interface
    {
    }
```

```
    [Guid("7BD20046-DF8C-44A6-8F6B-687FAA26FA71"),
        InterfaceType(ComInterfaceType.InterfaceIsIDispatch)]
    public interface ComClass1Events
    {
    }

    [Guid("0D53A3E8-E51A-49C7-944E-E72A2064F938"),
        ClassInterface(ClassInterfaceType.None),
        ComSourceInterfaces(typeof(ComClass1Events))]
    public class ComClass1 : ComClass1Interface
    {
    }
}
```

See also

- Interoperability
- Build Page, Project Designer (C#)

Walkthrough: Office Programming in C#

Article • 03/01/2023

C# offers features that improve Microsoft Office programming. Helpful C# features include named and optional arguments and return values of type `dynamic`. In COM programming, you can omit the `ref` keyword and gain access to indexed properties.

Both languages enable embedding of type information, which allows deployment of assemblies that interact with COM components without deploying primary interop assemblies (PIAs) to the user's computer. For more information, see Walkthrough: Embedding Types from Managed Assemblies.

This walkthrough demonstrates these features in the context of Office programming, but many of these features are also useful in general programming. In the walkthrough, you use an Excel Add-in application to create an Excel workbook. Next, you create a Word document that contains a link to the workbook. Finally, you see how to enable and disable the PIA dependency.

> **Important**
>
> VSTO (Visual Studio Tools for Office) relies on the .NET Framework. COM add-ins can also be written with the .NET Framework. Office Add-ins cannot be created with .NET Core and .NET 5+, the latest versions of .NET. This is because .NET Core/.NET 5+ cannot work together with .NET Framework in the same process and may lead to add-in load failures. You can continue to use .NET Framework to write VSTO and COM add-ins for Office. Microsoft will not be updating VSTO or the COM add-in platform to use .NET Core or .NET 5+. You can take advantage of .NET Core and .NET 5+, including ASP.NET Core, to create the server side of Office Web Add-ins.

Prerequisites

You must have Microsoft Office Excel and Microsoft Office Word installed on your computer to complete this walkthrough.

> **Note**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio

edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

Set up an Excel Add-in application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then select **Project**.
3. In the **Installed Templates** pane, expand **C#**, expand **Office**, and then select the version year of the Office product.
4. In the **Templates** pane, select **Excel <version> Add-in**.
5. Look at the top of the **Templates** pane to make sure that **.NET Framework 4**, or a later version, appears in the **Target Framework** box.
6. Type a name for your project in the **Name** box, if you want to.
7. Select **OK**.
8. The new project appears in **Solution Explorer**.

Add references

1. In **Solution Explorer**, right-click your project's name and then select **Add Reference**. The **Add Reference** dialog box appears.
2. On the **Assemblies** tab, select **Microsoft.Office.Interop.Excel**, version `<version>.0.0.0` (for a key to the Office product version numbers, see Microsoft Versions), in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Word**, `version <version>.0.0.0`. If you don't see the assemblies, you may need to install them (see How to: Install Office Primary Interop Assemblies).
3. Select **OK**.

Add necessary Imports statements or using directives

In **Solution Explorer**, right-click the **ThisAddIn.cs** file and then select **View Code**. Add the following `using` directives (C#) to the top of the code file if they aren't already present.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
```

```
using Word = Microsoft.Office.Interop.Word;
```

Create a list of bank accounts

In **Solution Explorer**, right-click your project's name, select **Add**, and then select **Class**. Name the class Account.cs. Select **Add**. Replace the definition of the `Account` class with the following code. The class definitions use *automatically implemented properties*.

C#

```
class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

To create a `bankAccounts` list that contains two accounts, add the following code to the `ThisAddIn_Startup` method in *ThisAddIn.cs*. The list declarations use *collection initializers*.

C#

```
var bankAccounts = new List<Account>
{
    new Account
    {
        ID = 345,
        Balance = 541.27
    },
    new Account
    {
        ID = 123,
        Balance = -127.44
    }
};
```

Export data to Excel

In the same file, add the following method to the `ThisAddIn` class. The method sets up an Excel workbook and exports data to it.

C#

```
void DisplayInExcel(IEnumerable<Account> accounts,
          Action<Account, Excel.Range> DisplayFunc)
```

```
{
    var excelApp = this.Application;
    // Add a new Excel workbook.
    excelApp.Workbooks.Add();
    excelApp.Visible = true;
    excelApp.Range["A1"].Value = "ID";
    excelApp.Range["B1"].Value = "Balance";
    excelApp.Range["A2"].Select();

    foreach (var ac in accounts)
    {
        DisplayFunc(ac, excelApp.ActiveCell);
        excelApp.ActiveCell.Offset[1, 0].Select();
    }
    // Copy the results to the Clipboard.
    excelApp.Range["A1:B3"].Copy();
}
```

- Method Add has an *optional parameter* for specifying a particular template. Optional parameters enable you to omit the argument for that parameter if you want to use the parameter's default value. Because the previous example has no arguments, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `excelApp.Workbooks.Add(Type.Missing)`. For more information, see Named and Optional Arguments.
- The `Range` and `Offset` properties of the Range object use the *indexed properties* feature. This feature enables you to consume these properties from COM types by using the following typical C# syntax. Indexed properties also enable you to use the `Value` property of the `Range` object, eliminating the need to use the `Value2` property. The `Value` property is indexed, but the index is optional. Optional arguments and indexed properties work together in the following example.

C#

```
// Visual C# 2010 provides indexed properties for COM programming.
excelApp.Range["A1"].Value = "ID";
excelApp.ActiveCell.Offset[1, 0].Select();
```

You can't create indexed properties of your own. The feature only supports consumption of existing indexed properties.

Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

C#

```
excelApp.Columns[1].AutoFit();
excelApp.Columns[2].AutoFit();
```

These additions demonstrate another feature in C#: treating `Object` values returned from COM hosts such as Office as if they have type dynamic. COM objects are treated as `dynamic` automatically when **Embed Interop Types** has its default value, `True`, or, equivalently, when you reference the assembly with the **EmbedInteropTypes** compiler option. For more information about embedding interop types, see procedures "To find the PIA reference" and "To restore the PIA dependency" later in this article. For more information about `dynamic`, see dynamic or Using Type dynamic.

Invoke DisplayInExcel

Add the following code at the end of the `ThisAddIn_StartUp` method. The call to `DisplayInExcel` contains two arguments. The first argument is the name of the list of accounts processed. The second argument is a multiline lambda expression defining how to process the data. The `ID` and `balance` values for each account are displayed in adjacent cells, and the row is displayed in red if the balance is less than zero. For more information, see Lambda Expressions.

C#

```
DisplayInExcel(bankAccounts, (account, cell) =>
// This multiline lambda expression sets custom processing rules
// for the bankAccounts.
{
    cell.Value = account.ID;
    cell.Offset[0, 1].Value = account.Balance;
    if (account.Balance < 0)
    {
        cell.Interior.Color = 255;
        cell.Offset[0, 1].Interior.Color = 255;
    }
});
```

To run the program, press F5. An Excel worksheet appears that contains the data from the accounts.

Add a Word document

Add the following code at the end of the `ThisAddIn_StartUp` method to create a Word document that contains a link to the Excel workbook.

C#

```
var wordApp = new Word.Application();
wordApp.Visible = true;
wordApp.Documents.Add();
wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
```

This code demonstrates several of the features in C#: the ability to omit the `ref` keyword in COM programming, named arguments, and optional arguments.The PasteSpecial method has seven parameters, all of which are optional reference parameters. Named and optional arguments enable you to designate the parameters you want to access by name and to send arguments to only those parameters. In this example, arguments indicate creating a link to the workbook on the Clipboard (parameter `Link`) and displaying that the link in the Word document as an icon (parameter `DisplayAsIcon`). C# also enables you to omit the `ref` keyword for these arguments.

Run the application

Press F5 to run the application. Excel starts and displays a table that contains the information from the two accounts in `bankAccounts`. Then a Word document appears that contains a link to the Excel table.

Clean up the completed project

In Visual Studio, select **Clean Solution** on the **Build** menu. Otherwise, the add-in runs every time that you open Excel on your computer.

Find the PIA reference

1. Run the application again, but don't select **Clean Solution**.
2. Select the **Start**. Locate **Microsoft Visual Studio <version>** and open a developer command prompt.
3. Type `ildasm` in the Developer Command Prompt for Visual Studio window, and then press `Enter`. The IL DASM window appears.
4. On the **File** menu in the IL DASM window, select **File** > **Open**. Double-click **Visual Studio <version>**, and then double-click **Projects**. Open the folder for your project, and look in the bin/Debug folder for *your project name*.dll. Double-click *your project name*.dll. A new window displays your project's attributes, in addition to references to other modules and assemblies. The assembly includes the

namespaces `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word`. By default in Visual Studio, the compiler imports the types you need from a referenced PIA into your assembly. For more information, see How to: View Assembly Contents.

5. Double-click the **MANIFEST** icon. A window appears that contains a list of assemblies that contain items referenced by the project. `Microsoft.Office.Interop.Excel` and `Microsoft.Office.Interop.Word` aren't in the list. Because you imported the types your project needs into your assembly, you aren't required to install references to a PIA. Importing the types into your assembly makes deployment easier. The PIAs don't have to be present on the user's computer. An application doesn't require deployment of a specific version of a PIA. Applications can work with multiple versions of Office, provided that the necessary APIs exist in all versions. Because deployment of PIAs is no longer necessary, you can create an application in advanced scenarios that works with multiple versions of Office, including earlier versions. Your code can't use any APIs that aren't available in the version of Office you're working with. It isn't always clear whether a particular API was available in an earlier version. Working with earlier versions of Office isn't recommended.
6. Close the manifest window and the assembly window.

Restore the PIA dependency

1. In **Solution Explorer**, select the **Show All Files** button. Expand the **References** folder and select **Microsoft.Office.Interop.Excel**. Press F4 to display the **Properties** window.
2. In the **Properties** window, change the **Embed Interop Types** property from **True** to **False**.
3. Repeat steps 1 and 2 in this procedure for `Microsoft.Office.Interop.Word`.
4. In C#, comment out the two calls to `Autofit` at the end of the `DisplayInExcel` method.
5. Press F5 to verify that the project still runs correctly.
6. Repeat steps 1-3 from the previous procedure to open the assembly window. Notice that `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are no longer in the list of embedded assemblies.
7. Double-click the **MANIFEST** icon and scroll through the list of referenced assemblies. Both `Microsoft.Office.Interop.Word` and `Microsoft.Office.Interop.Excel` are in the list. Because the application references the Excel and Word PIAs, and the **Embed Interop Types** property is **False**, both assemblies must exist on the end user's computer.

8. In Visual Studio, select **Clean Solution** on the **Build** menu to clean up the completed project.

See also

- Automatically implemented properties (C#)
- Object and collection initializers
- Visual Studio Tools for Office (VSTO)
- Named and optional arguments
- dynamic
- Using type dynamic
- Lambda expressions (C#)
- Walkthrough: Embedding type information from Microsoft Office assemblies in Visual Studio
- Walkthrough: Embedding types from managed assemblies
- Walkthrough: Creating your first VSTO add-in for Excel

How to use platform invoke to play a WAV file

Article • 02/25/2023

The following C# code example illustrates how to use platform invoke services to play a WAV sound file on the Windows operating system.

Example

This example code uses DllImportAttribute to import `winmm.dll`'s `PlaySound` method entry point as `Form1 PlaySound()`. The example has a simple Windows Form with a button. Clicking the button opens a standard windows OpenFileDialog dialog box so that you can open a file to play. When a wave file is selected, it's played by using the `PlaySound()` method of the *winmm.dll* library. For more information about this method, see Using the PlaySound function with Waveform-Audio Files. Browse and select a file that has a .wav extension, and then select **Open** to play the wave file by using platform invoke. A text box shows the full path of the file selected.

C#

```
using System.Runtime.InteropServices;

namespace WinSound;

public partial class Form1 : Form
{
    private TextBox textBox1;
    private Button button1;

    public Form1()  // Constructor.
    {
        InitializeComponent();
    }

    [DllImport("winmm.DLL", EntryPoint = "PlaySound", SetLastError = true,
CharSet = CharSet.Unicode, ThrowOnUnmappableChar = true)]
    private static extern bool PlaySound(string szSound, System.IntPtr hMod,
PlaySoundFlags flags);

    [System.Flags]
    public enum PlaySoundFlags : int
    {
        SND_SYNC = 0x0000,
        SND_ASYNC = 0x0001,
        SND_NODEFAULT = 0x0002,
```

```
        SND_LOOP = 0x0008,
        SND_NOSTOP = 0x0010,
        SND_NOWAIT = 0x00002000,
        SND_FILENAME = 0x00020000,
        SND_RESOURCE = 0x00040004
    }

    private void button1_Click(object sender, System.EventArgs e)
    {
        var dialog1 = new OpenFileDialog();

        dialog1.Title = "Browse to find sound file to play";
        dialog1.InitialDirectory = @"c:\";
        //<Snippet5>
        dialog1.Filter = "Wav Files (*.wav)|*.wav";
        //</Snippet5>
        dialog1.FilterIndex = 2;
        dialog1.RestoreDirectory = true;

        if (dialog1.ShowDialog() == DialogResult.OK)
        {
            textBox1.Text = dialog1.FileName;
            PlaySound(dialog1.FileName, new System.IntPtr(),
PlaySoundFlags.SND_SYNC);
        }
    }

    private void Form1_Load(object sender, EventArgs e)
    {
        // Including this empty method in the sample because in the IDE,
        // when users click on the form, generates code that looks for a
default method
        // with this name. We add it here to prevent confusion for those
using the samples.
    }
}
```

The **Open Files** dialog box is filtered to show only files that have a .wav extension through the filter settings.

Compiling the code

Create a new C# Windows Forms Application project in Visual Studio and name it **WinSound**. Copy the preceding code, and paste it over the contents of the *Form1.cs* file. Copy the following code, and paste it in the *Form1.Designer.cs* file, in the `InitializeComponent()` method, after any existing code.

```
C#
```

```
this.button1 = new System.Windows.Forms.Button();
this.textBox1 = new System.Windows.Forms.TextBox();
this.SuspendLayout();
//
// button1
//
this.button1.Location = new System.Drawing.Point(192, 40);
this.button1.Name = "button1";
this.button1.Size = new System.Drawing.Size(88, 24);
this.button1.TabIndex = 0;
this.button1.Text = "Browse";
this.button1.Click += new System.EventHandler(this.button1_Click);
//
// textBox1
//
this.textBox1.Location = new System.Drawing.Point(8, 40);
this.textBox1.Name = "textBox1";
this.textBox1.Size = new System.Drawing.Size(168, 20);
this.textBox1.TabIndex = 1;
this.textBox1.Text = "File path";
//
// Form1
//
this.AutoScaleDimensions = new System.Drawing.SizeF(5, 13);
this.ClientSize = new System.Drawing.Size(292, 266);
this.Controls.Add(this.textBox1);
this.Controls.Add(this.button1);
this.Name = "Form1";
this.Text = "Platform Invoke WinSound C#";
this.ResumeLayout(false);
this.PerformLayout();
```

Compile and run the code.

See also

- A Closer Look at Platform Invoke
- Marshaling Data with Platform Invoke

How to use indexed properties in COM interop programming

Article • 03/01/2023

Indexed properties work together with other features in C#, such as named and optional arguments, a new type (dynamic), and embedded type information, to enhance Microsoft Office programming.

> **Important**
>
> VSTO (Visual Studio Tools for Office) relies on the .NET Framework. COM add-ins can also be written with the .NET Framework. Office Add-ins cannot be created with .NET Core and .NET 5+, the latest versions of .NET. This is because .NET Core/.NET 5+ cannot work together with .NET Framework in the same process and may lead to add-in load failures. You can continue to use .NET Framework to write VSTO and COM add-ins for Office. Microsoft will not be updating VSTO or the COM add-in platform to use .NET Core or .NET 5+. You can take advantage of .NET Core and .NET 5+, including ASP.NET Core, to create the server side of Office Web Add-ins.

In earlier versions of C#, methods are accessible as properties only if the `get` method has no parameters and the `set` method has one and only one value parameter. However, not all COM properties meet those restrictions. For example, the Excel Range[] property has a `get` accessor that requires a parameter for the name of the range. In the past, because you couldn't access the `Range` property directly, you had to use the `get_Range` method instead, as shown in the following example.

C#

```
// Visual C# 2008 and earlier.
var excelApp = new Excel.Application();
// . . .
Excel.Range targetRange = excelApp.get_Range("A1", Type.Missing);
```

Indexed properties enable you to write the following instead:

C#

```
// Visual C# 2010.
var excelApp = new Excel.Application();
```

```
// . . .
Excel.Range targetRange = excelApp.Range["A1"];
```

The previous example also uses the optional arguments feature, which enables you to omit `Type.Missing`.

Indexed properties enable you to write the following code.

C#

```
// Visual C# 2010.
targetRange.Value = "Name";
```

You can't create indexed properties of your own. The feature only supports consumption of existing indexed properties.

Example

The following code shows a complete example. For more information about how to set up a project that accesses the Office API, see How to access Office interop objects by using C# features.

C#

```
// You must add a reference to Microsoft.Office.Interop.Excel to run
// this example.
using System;
using Excel = Microsoft.Office.Interop.Excel;

namespace IndexedProperties
{
    class Program
    {
        static void Main(string[] args)
        {
            CSharp2010();
        }

        static void CSharp2010()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add();
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.Range["A1"];
            targetRange.Value = "Name";
        }
```

```
        static void CSharp2008()
        {
            var excelApp = new Excel.Application();
            excelApp.Workbooks.Add(Type.Missing);
            excelApp.Visible = true;

            Excel.Range targetRange = excelApp.get_Range("A1",
Type.Missing);
            targetRange.set_Value(Type.Missing, "Name");
            // Or
            //targetRange.Value2 = "Name";
        }
    }
}
```

See also

- Named and Optional Arguments
- dynamic
- Using Type dynamic

How to access Office interop objects

Article • 03/01/2023

C# has features that simplify access to Office API objects. The new features include named and optional arguments, a new type called `dynamic`, and the ability to pass arguments to reference parameters in COM methods as if they were value parameters.

In this article, you use the new features to write code that creates and displays a Microsoft Office Excel worksheet. You write code to add an Office Word document that contains an icon that is linked to the Excel worksheet.

To complete this walkthrough, you must have Microsoft Office Excel 2007 and Microsoft Office Word 2007, or later versions, installed on your computer.

> **Note**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

> **Important**
>
> VSTO (Visual Studio Tools for Office) relies on the .NET Framework. COM add-ins can also be written with the .NET Framework. Office Add-ins cannot be created with .NET Core and .NET 5+, the latest versions of .NET. This is because .NET Core/.NET 5+ cannot work together with .NET Framework in the same process and may lead to add-in load failures. You can continue to use .NET Framework to write VSTO and COM add-ins for Office. Microsoft will not be updating VSTO or the COM add-in platform to use .NET Core or .NET 5+. You can take advantage of .NET Core and .NET 5+, including ASP.NET Core, to create the server side of Office Web Add-ins.

To create a new console application

1. Start Visual Studio.
2. On the **File** menu, point to **New**, and then select **Project**. The **New Project** dialog box appears.
3. In the **Installed Templates** pane, expand **C#**, and then select **Windows**.

4. Look at the top of the **New Project** dialog box to make sure to select **.NET Framework 4** (or later version) as a target framework.
5. In the **Templates** pane, select **Console Application**.
6. Type a name for your project in the **Name** field.
7. Select **OK**.

The new project appears in **Solution Explorer**.

To add references

1. In **Solution Explorer**, right-click your project's name and then select **Add Reference**. The **Add Reference** dialog box appears.
2. On the **Assemblies** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list, and then hold down the CTRL key and select **Microsoft.Office.Interop.Excel**. If you don't see the assemblies, you may need to install them. See How to: Install Office Primary Interop Assemblies.
3. Select **OK**.

To add necessary using directives

In **Solution Explorer**, right-click the *Program.cs* file and then select **View Code**. Add the following `using` directives to the top of the code file:

```
C#

using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;
```

To create a list of bank accounts

Paste the following class definition into **Program.cs**, under the `Program` class.

```
C#

public class Account
{
    public int ID { get; set; }
    public double Balance { get; set; }
}
```

Add the following code to the `Main` method to create a `bankAccounts` list that contains two accounts.

C#

```
// Create a list of accounts.
var bankAccounts = new List<Account> {
    new Account {
                  ID = 345678,
                  Balance = 541.27
                },
    new Account {
                  ID = 1230221,
                  Balance = -127.44
                }
};
```

To declare a method that exports account information to Excel

1. Add the following method to the `Program` class to set up an Excel worksheet. Method Add has an optional parameter for specifying a particular template. Optional parameters enable you to omit the argument for that parameter if you want to use the parameter's default value. Because you didn't supply an argument, `Add` uses the default template and creates a new workbook. The equivalent statement in earlier versions of C# requires a placeholder argument: `ExcelApp.Workbooks.Add(Type.Missing)`.

C#

```
static void DisplayInExcel(IEnumerable<Account> accounts)
{
    var excelApp = new Excel.Application();
    // Make the object visible.
    excelApp.Visible = true;

    // Create a new, empty workbook and add it to the collection returned
    // by property Workbooks. The new workbook becomes the active workbook.
    // Add has an optional parameter for specifying a particular template.
    // Because no argument is sent in this example, Add creates a new
workbook.
    excelApp.Workbooks.Add();

    // This example uses a single workSheet. The explicit type casting is
    // removed in a later procedure.
```

```
    Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet;
}
```

Add the following code at the end of `DisplayInExcel`. The code inserts values into the first two columns of the first row of the worksheet.

C#

```
// Establish column headings in cells A1 and B1.
workSheet.Cells[1, "A"] = "ID Number";
workSheet.Cells[1, "B"] = "Current Balance";
```

Add the following code at the end of `DisplayInExcel`. The `foreach` loop puts the information from the list of accounts into the first two columns of successive rows of the worksheet.

C#

```
var row = 1;
foreach (var acct in accounts)
{
    row++;
    workSheet.Cells[row, "A"] = acct.ID;
    workSheet.Cells[row, "B"] = acct.Balance;
}
```

Add the following code at the end of `DisplayInExcel` to adjust the column widths to fit the content.

C#

```
workSheet.Columns[1].AutoFit();
workSheet.Columns[2].AutoFit();
```

Earlier versions of C# require explicit casting for these operations because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel Range method. The following lines show the casting.

C#

```
((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

C# converts the returned `Object` to `dynamic` automatically if the assembly is referenced by the EmbedInteropTypes compiler option or, equivalently, if the Excel **Embed Interop Types** property is true. True is the default value for this property.

To run the project

Add the following line at the end of `Main`.

```
C#

// Display the list in an Excel spreadsheet.
DisplayInExcel(bankAccounts);
```

Press CTRL+F5. An Excel worksheet appears that contains the data from the two accounts.

To add a Word document

The following code opens a Word application and creates an icon that links to the Excel worksheet. Paste method `CreateIconInWordDoc`, provided later in this step, into the `Program` class. `CreateIconInWordDoc` uses named and optional arguments to reduce the complexity of the method calls to Add and PasteSpecial. These calls incorporate two other features that simplify calls to COM methods that have reference parameters. First, you can send arguments to the reference parameters as if they were value parameters. That is, you can send values directly, without creating a variable for each reference parameter. The compiler generates temporary variables to hold the argument values, and discards the variables when you return from the call. Second, you can omit the `ref` keyword in the argument list.

The `Add` method has four reference parameters, all of which are optional. You can omit arguments for any or all of the parameters if you want to use their default values.

The `PasteSpecial` method inserts the contents of the Clipboard. The method has seven reference parameters, all of which are optional. The following code specifies arguments for two of them: `Link`, to create a link to the source of the Clipboard contents, and `DisplayAsIcon`, to display the link as an icon. You can use named arguments for those two arguments and omit the others. Although these arguments are reference parameters, you don't have to use the `ref` keyword, or to create variables to send in as arguments. You can send the values directly.

```
C#
```

```
static void CreateIconInWordDoc()
{
    var wordApp = new Word.Application();
    wordApp.Visible = true;

    // The Add method has four reference parameters, all of which are
    // optional. Visual C# allows you to omit arguments for them if
    // the default values are what you want.
    wordApp.Documents.Add();

    // PasteSpecial has seven reference parameters, all of which are
    // optional. This example uses named arguments to specify values
    // for two of the parameters. Although these are reference
    // parameters, you do not need to use the ref keyword, or to create
    // variables to send in as arguments. You can send the values directly.
    wordApp.Selection.PasteSpecial( Link: true, DisplayAsIcon: true);
}
```

Add the following statement at the end of `Main`.

C#

```
// Create a Word document that contains an icon that links to
// the spreadsheet.
CreateIconInWordDoc();
```

Add the following statement at the end of `DisplayInExcel`. The `Copy` method adds the worksheet to the Clipboard.

C#

```
// Put the spreadsheet contents on the clipboard. The Copy method has one
// optional parameter for specifying a destination. Because no argument
// is sent, the destination is the Clipboard.
workSheet.Range["A1:B3"].Copy();
```

Press CTRL+F5. A Word document appears that contains an icon. Double-click the icon to bring the worksheet to the foreground.

To set the Embed Interop Types property

More enhancements are possible when you call a COM type that doesn't require a primary interop assembly (PIA) at run time. Removing the dependency on PIAs results in version independence and easier deployment. For more information about the

advantages of programming without PIAs, see Walkthrough: Embedding Types from Managed Assemblies.

In addition, programming is easier because the `dynamic` type represents the required and returned types declared in COM methods. Variables that have type `dynamic` aren't evaluated until run time, which eliminates the need for explicit casting. For more information, see Using Type dynamic.

Embedding type information instead of using PIAs is default behavior. Because of that default, several of the previous examples are simplified. You don't need any explicit casting. For example, the declaration of `worksheet` in `DisplayInExcel` is written as `Excel._Worksheet workSheet = excelApp.ActiveSheet` rather than `Excel._Worksheet workSheet = (Excel.Worksheet)excelApp.ActiveSheet`. The calls to `AutoFit` in the same method also would require explicit casting without the default, because `ExcelApp.Columns[1]` returns an `Object`, and `AutoFit` is an Excel method. The following code shows the casting.

```
C#

((Excel.Range)workSheet.Columns[1]).AutoFit();
((Excel.Range)workSheet.Columns[2]).AutoFit();
```

To change the default and use PIAs instead of embedding type information, expand the References node in Solution Explorer, and then select **Microsoft.Office.Interop.Excel** or **Microsoft.Office.Interop.Word**. If you can't see the **Properties** window, press **F4**. Find **Embed Interop Types** in the list of properties, and change its value to **False**. Equivalently, you can compile by using the **References** compiler option instead of **EmbedInteropTypes** at a command prompt.

To add additional formatting to the table

Replace the two calls to `AutoFit` in `DisplayInExcel` with the following statement.

```
C#

// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

The AutoFormat method has seven value parameters, all of which are optional. Named and optional arguments enable you to provide arguments for none, some, or all of them. In the previous statement, you supply an argument for only one of the

parameters, `Format`. Because `Format` is the first parameter in the parameter list, you don't have to provide the parameter name. However, the statement might be easier to understand if you include the parameter name, as shown in the following code.

C#

```
// Call to AutoFormat in Visual C# 2010.
workSheet.Range["A1", "B3"].AutoFormat(Format:
    Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);
```

Press CTRL+F5 to see the result. You can find other formats in the listed in the XlRangeAutoFormat enumeration.

Example

The following code shows the complete example.

C#

```
using System.Collections.Generic;
using Excel = Microsoft.Office.Interop.Excel;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeProgrammingWalkthruComplete
{
    class Walkthrough
    {
        static void Main(string[] args)
        {
            // Create a list of accounts.
            var bankAccounts = new List<Account>
            {
                new Account {
                              ID = 345678,
                              Balance = 541.27
                            },
                new Account {
                              ID = 1230221,
                              Balance = -127.44
                            }
            };

            // Display the list in an Excel spreadsheet.
            DisplayInExcel(bankAccounts);

            // Create a Word document that contains an icon that links to
            // the spreadsheet.
            CreateIconInWordDoc();
        }
```

```
        static void DisplayInExcel(IEnumerable<Account> accounts)
        {
            var excelApp = new Excel.Application();
            // Make the object visible.
            excelApp.Visible = true;

            // Create a new, empty workbook and add it to the collection
returned
            // by property Workbooks. The new workbook becomes the active
workbook.
            // Add has an optional parameter for specifying a particular
template.
            // Because no argument is sent in this example, Add creates a
new workbook.
            excelApp.Workbooks.Add();

            // This example uses a single workSheet.
            Excel._Worksheet workSheet = excelApp.ActiveSheet;

            // Earlier versions of C# require explicit casting.
            //Excel._Worksheet workSheet =
(Excel.Worksheet)excelApp.ActiveSheet;

            // Establish column headings in cells A1 and B1.
            workSheet.Cells[1, "A"] = "ID Number";
            workSheet.Cells[1, "B"] = "Current Balance";

            var row = 1;
            foreach (var acct in accounts)
            {
                row++;
                workSheet.Cells[row, "A"] = acct.ID;
                workSheet.Cells[row, "B"] = acct.Balance;
            }

            workSheet.Columns[1].AutoFit();
            workSheet.Columns[2].AutoFit();

            // Call to AutoFormat in Visual C#. This statement replaces the
            // two calls to AutoFit.
            workSheet.Range["A1", "B3"].AutoFormat(
                Excel.XlRangeAutoFormat.xlRangeAutoFormatClassic2);

            // Put the spreadsheet contents on the clipboard. The Copy
method has one
            // optional parameter for specifying a destination. Because no
argument
            // is sent, the destination is the Clipboard.
            workSheet.Range["A1:B3"].Copy();
        }

        static void CreateIconInWordDoc()
        {
            var wordApp = new Word.Application();
```

```
            wordApp.Visible = true;

            // The Add method has four reference parameters, all of which
are
            // optional. Visual C# allows you to omit arguments for them if
            // the default values are what you want.
            wordApp.Documents.Add();

            // PasteSpecial has seven reference parameters, all of which are
            // optional. This example uses named arguments to specify values
            // for two of the parameters. Although these are reference
            // parameters, you do not need to use the ref keyword, or to
create
            // variables to send in as arguments. You can send the values
directly.
            wordApp.Selection.PasteSpecial(Link: true, DisplayAsIcon: true);
        }
    }

    public class Account
    {
        public int ID { get; set; }
        public double Balance { get; set; }
    }
}
```

See also

- Type.Missing
- dynamic
- Named and Optional Arguments
- How to use named and optional arguments in Office programming

How to use named and optional arguments in Office programming

07/03/2025

Named arguments and optional arguments enhance convenience, flexibility, and readability in C# programming. In addition, these features greatly facilitate access to COM interfaces such as the Microsoft Office automation APIs.

> **Important**
>
> VSTO (Visual Studio Tools for Office) relies on the .NET Framework. COM add-ins can also be written with the .NET Framework. Office Add-ins cannot be created with .NET Core and .NET 5+, the latest versions of .NET. This is because .NET Core/.NET 5+ cannot work together with .NET Framework in the same process and may lead to add-in load failures. You can continue to use .NET Framework to write VSTO and COM add-ins for Office. Microsoft will not be updating VSTO or the COM add-in platform to use .NET Core or .NET 5+. You can take advantage of .NET Core and .NET 5+, including ASP.NET Core, to create the server side of Office Web Add-ins.

In the following example, method ConvertToTable has 16 parameters that represent characteristics of a table, such as number of columns and rows, formatting, borders, fonts, and colors. All 16 parameters are optional, because most of the time you don't want to specify particular values for all of them. However, without named and optional arguments, you must provide a value or a placeholder value. With named and optional arguments, you specify values only for the parameters required for your project.

You must have Microsoft Office Word installed on your computer to complete these procedures.

> **Note**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

Create a new console application

Start Visual Studio. On the **File** menu, point to **New**, and then select **Project**. In the **Templates Categories** pane, expand **C#**, and then select **Windows**. Look in the top of the **Templates** pane to make sure that **.NET Framework 4** appears in the **Target Framework** box. In the **Templates** pane, select **Console Application**. Type a name for your project in the **Name** field. Select **OK**. The new project appears in **Solution Explorer**.

Add a reference

In **Solution Explorer**, right-click your project's name and then select **Add Reference**. The **Add Reference** dialog box appears. On the **.NET** page, select **Microsoft.Office.Interop.Word** in the **Component Name** list. Select **OK**.

Add necessary using directives

In **Solution Explorer**, right-click the *Program.cs* file and then select **View Code**. Add the following `using` directives to the top of the code file:

C#

```
using Word = Microsoft.Office.Interop.Word;
```

Display text in a Word document

In the `Program` class in *Program.cs*, add the following method to create a Word application and a Word document. The Add method has four optional parameters. This example uses their default values. Therefore, no arguments are necessary in the calling statement.

> ⓘ **Note**
>
> To avoid COM threading and timing issues that can cause exceptions like "The message filter indicated that the application is busy" (HRESULT 0x8001010A), the Word application is kept invisible during operations and only made visible after all operations are complete.

C#

```
static void DisplayInWord()
{
    var wordApp = new Word.Application();
    // Keep Word invisible during operations to avoid COM threading issues
    wordApp.Visible = false;
    // docs is a collection of all the Document objects currently
```

```
    // open in Word.
    Word.Documents docs = wordApp.Documents;

    // Add a document to the collection and name it doc.
    Word.Document doc = docs.Add();

    // Make Word visible after operations are complete
    wordApp.Visible = true;
}
```

Add the following code at the end of the method to define where to display text in the document, and what text to display:

C#

```
// Define a range, a contiguous area in the document, by specifying
// a starting and ending character position. Currently, the document
// is empty.
Word.Range range = doc.Range(0, 0);

// Use the InsertAfter method to insert a string at the end of the
// current range.
range.InsertAfter("Testing, testing, testing. . .");
```

Run the application

Add the following statement to Main:

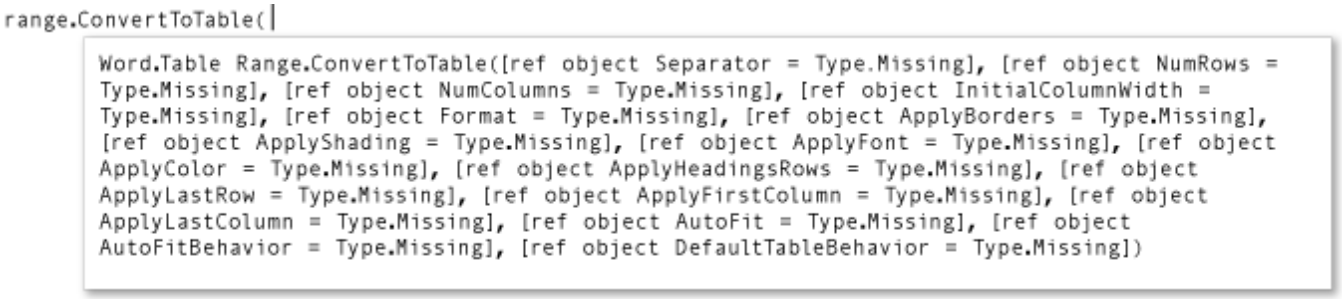
C#

```
DisplayInWord();
```

Press `CTRL` + `F5` to run the project. A Word document appears that contains the specified text.

Change the text to a table

Use the `ConvertToTable` method to enclose the text in a table. The method has 16 optional parameters. IntelliSense encloses optional parameters in brackets, as shown in the following illustration. The default values of `Type.Missing` are the simple name for `System.Type.Missing`.



Named and optional arguments enable you to specify values for only the parameters that you want to change. Add the following code to the end of method `DisplayInWord` to create a table. The argument specifies that the commas in the text string in `range` separate the cells of the table.

C#

```
// Convert to a simple table. The table will have a single row with
// three columns.
range.ConvertToTable(Separator: ",");
```

Press `CTRL` + `F5` to run the project.

Experiment with other parameters

Change the table so that it has one column and three rows, replace the last line in `DisplayInWord` with the following statement and then type `CTRL` + `F5`.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);
```

Specify a predefined format for the table, replace the last line in `DisplayInWord` with the following statement and then type `CTRL` + `F5`. The format can be any of the WdTableFormat constants.

C#

```
range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
    Format: Word.WdTableFormat.wdTableFormatElegant);
```

Example

The following code includes the full example:

C#

```
using System;
using Word = Microsoft.Office.Interop.Word;

namespace OfficeHowTo
{
    class WordProgram
    {
        static void Main(string[] args)
        {
            DisplayInWord();
        }

        static void DisplayInWord()
        {
            var wordApp = new Word.Application();
            // Keep Word invisible during operations to avoid COM threading issues
            wordApp.Visible = false;
            // docs is a collection of all the Document objects currently
            // open in Word.
            Word.Documents docs = wordApp.Documents;

            // Add a document to the collection and name it doc.
            Word.Document doc = docs.Add();

            // Define a range, a contiguous area in the document, by specifying
            // a starting and ending character position. Currently, the document
            // is empty.
            Word.Range range = doc.Range(0, 0);

            // Use the InsertAfter method to insert a string at the end of the
            // current range.
            range.InsertAfter("Testing, testing, testing. . .");

            // You can comment out any or all of the following statements to
            // see the effect of each one in the Word document.

            // Next, use the ConvertToTable method to put the text into a table.
            // The method has 16 optional parameters. You only have to specify
            // values for those you want to change.

            // Convert to a simple table. The table will have a single row with
            // three columns.
            range.ConvertToTable(Separator: ",");

            // Change to a single column with three rows..
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1);

            // Format the table.
            range.ConvertToTable(Separator: ",", AutoFit: true, NumColumns: 1,
                Format: Word.WdTableFormat.wdTableFormatElegant);

            // Make Word visible after all operations are complete
```

```
            wordApp.Visible = true;
        }
    }
}
```

Using type dynamic

Article • 02/25/2023

The `dynamic` type is a static type, but an object of type `dynamic` bypasses static type checking. In most cases, it functions like it has type `object`. The compiler assumes a `dynamic` element supports any operation. Therefore, you don't have to determine whether the object gets its value from a COM API, from a dynamic language such as IronPython, from the HTML Document Object Model (DOM), from reflection, or from somewhere else in the program. However, if the code isn't valid, errors surface at run time.

For example, if instance method `exampleMethod1` in the following code has only one parameter, the compiler recognizes that the first call to the method, `ec.exampleMethod1(10, 4)`, isn't valid because it contains two arguments. The call causes a compiler error. The compiler doesn't check the second call to the method, `dynamic_ec.exampleMethod1(10, 4)`, because the type of `dynamic_ec` is `dynamic`. Therefore, no compiler error is reported. However, the error doesn't escape notice indefinitely. It appears at run time and causes a run-time exception.

C#

```
static void Main(string[] args)
{
    ExampleClass ec = new ExampleClass();
    // The following call to exampleMethod1 causes a compiler error
    // if exampleMethod1 has only one parameter. Uncomment the line
    // to see the error.
    //ec.exampleMethod1(10, 4);

    dynamic dynamic_ec = new ExampleClass();
    // The following line is not identified as an error by the
    // compiler, but it causes a run-time exception.
    dynamic_ec.exampleMethod1(10, 4);

    // The following calls also do not cause compiler errors, whether
    // appropriate methods exist or not.
    dynamic_ec.someMethod("some argument", 7, null);
    dynamic_ec.nonexistentMethod();
}
```

C#

```
class ExampleClass
{
    public ExampleClass() { }
```

```
    public ExampleClass(int v) { }

    public void exampleMethod1(int i) { }

    public void exampleMethod2(string str) { }
}
```

The role of the compiler in these examples is to package together information about what each statement is proposing to do to the `dynamic` object or expression. The runtime examines the stored information and any statement that isn't valid causes a run-time exception.

The result of most dynamic operations is itself `dynamic`. For example, if you rest the mouse pointer over the use of `testSum` in the following example, IntelliSense displays the type **(local variable) dynamic testSum**.

C#

```
dynamic d = 1;
var testSum = d + 3;
// Rest the mouse pointer over testSum in the following statement.
System.Console.WriteLine(testSum);
```

Operations in which the result isn't `dynamic` include:

- Conversions from `dynamic` to another type.
- Constructor calls that include arguments of type `dynamic`.

For example, the type of `testInstance` in the following declaration is `ExampleClass`, not `dynamic`:

C#

```
var testInstance = new ExampleClass(d);
```

Conversions

Conversions between dynamic objects and other types are easy. Conversions enable the developer to switch between dynamic and non-dynamic behavior.

You can convert any to `dynamic` implicitly, as shown in the following examples.

C#

```
dynamic d1 = 7;
dynamic d2 = "a string";
dynamic d3 = System.DateTime.Today;
dynamic d4 = System.Diagnostics.Process.GetProcesses();
```

Conversely, you can dynamically apply any implicit conversion to any expression of type `dynamic`.

C#

```
int i = d1;
string str = d2;
DateTime dt = d3;
System.Diagnostics.Process[] procs = d4;
```

Overload resolution with arguments of type dynamic

Overload resolution occurs at run time instead of at compile time if one or more of the arguments in a method call have the type `dynamic`, or if the receiver of the method call is of type `dynamic`. In the following example, if the only accessible `exampleMethod2` method takes a string argument, sending `d1` as the argument doesn't cause a compiler error, but it does cause a run-time exception. Overload resolution fails at run time because the run-time type of `d1` is `int`, and `exampleMethod2` requires a string.

C#

```
// Valid.
ec.exampleMethod2("a string");

// The following statement does not cause a compiler error, even though ec
is not
// dynamic. A run-time exception is raised because the run-time type of d1
is int.
ec.exampleMethod2(d1);
// The following statement does cause a compiler error.
//ec.exampleMethod2(7);
```

Dynamic language runtime

The dynamic language runtime (DLR) provides the infrastructure that supports the `dynamic` type in C#, and also the implementation of dynamic programming languages

such as IronPython and IronRuby. For more information about the DLR, see Dynamic Language Runtime Overview.

COM interop

Many COM methods allow for variation in argument types and return type by designating the types as `object`. COM interop necessitates explicit casting of the values to coordinate with strongly typed variables in C#. If you compile by using the **EmbedInteropTypes** (C# Compiler Options) option, the introduction of the `dynamic` type enables you to treat the occurrences of `object` in COM signatures as if they were of type `dynamic`, and thereby to avoid much of the casting. For more information on using the `dynamic` type with COM objects, see the article on How to access Office interop objects by using C# features.

Related articles

Expand table

Title	Description
dynamic	Describes the usage of the `dynamic` keyword.
Dynamic Language Runtime Overview	Provides an overview of the DLR, which is a runtime environment that adds a set of services for dynamic languages to the common language runtime (CLR).
Walkthrough: Creating and Using Dynamic Objects	Provides step-by-step instructions for creating a custom dynamic object and for creating a project that accesses an `IronPython` library.

Walkthrough: Creating and Using Dynamic Objects in C#

Article • 02/25/2023

Dynamic objects expose members such as properties and methods at run time, instead of at compile time. Dynamic objects enable you to create objects to work with structures that don't match a static type or format. For example, you can use a dynamic object to reference the HTML Document Object Model (DOM), which can contain any combination of valid HTML markup elements and attributes. Because each HTML document is unique, the members for a particular HTML document are determined at run time. A common method to reference an attribute of an HTML element is to pass the name of the attribute to the `GetProperty` method of the element. To reference the `id` attribute of the HTML element `<div id="Div1">`, you first obtain a reference to the `<div>` element, and then use `divElement.GetProperty("id")`. If you use a dynamic object, you can reference the `id` attribute as `divElement.id`.

Dynamic objects also provide convenient access to dynamic languages such as IronPython and IronRuby. You can use a dynamic object to refer to a dynamic script interpreted at run time.

You reference a dynamic object by using late binding. You specify the type of a late-bound object as `dynamic`.For more information, see dynamic.

You can create custom dynamic objects by using the classes in the System.Dynamic namespace. For example, you can create an ExpandoObject and specify the members of that object at run time. You can also create your own type that inherits the DynamicObject class. You can then override the members of the DynamicObject class to provide run-time dynamic functionality.

This article contains two independent walkthroughs:

- Create a custom object that dynamically exposes the contents of a text file as properties of an object.
- Create a project that uses an `IronPython` library.

Prerequisites

- Visual Studio 2022 version 17.3 or a later version with the **.NET desktop development** workload installed. The .NET 7 SDK is included when you select this workload.

> **Note**
>
> Your computer might show different names or locations for some of the Visual Studio user interface elements in the following instructions. The Visual Studio edition that you have and the settings that you use determine these elements. For more information, see Personalizing the IDE.

- For the second walkthrough, install IronPython for .NET. Go to their Download page to obtain the latest version.

Create a Custom Dynamic Object

The first walkthrough defines a custom dynamic object that searches the contents of a text file. A dynamic property specifies the text to search for. For example, if calling code specifies `dynamicFile.Sample`, the dynamic class returns a generic list of strings that contains all of the lines from the file that begin with "Sample". The search is case-insensitive. The dynamic class also supports two optional arguments. The first argument is a search option enum value that specifies that the dynamic class should search for matches at the start of the line, the end of the line, or anywhere in the line. The second argument specifies that the dynamic class should trim leading and trailing spaces from each line before searching. For example, if calling code specifies `dynamicFile.Sample(StringSearchOption.Contains)`, the dynamic class searches for "Sample" anywhere in a line. If calling code specifies `dynamicFile.Sample(StringSearchOption.StartsWith, false)`, the dynamic class searches for "Sample" at the start of each line, and doesn't remove leading and trailing spaces. The default behavior of the dynamic class is to search for a match at the start of each line and to remove leading and trailing spaces.

Create a custom dynamic class

Start Visual Studio. Select **Create a new project**. In the **Create a new project** dialog, select C#, select **Console Application**, and then select **Next**. In the **Configure your new project** dialog, enter `DynamicSample` for the **Project name**, and then select **Next**. In the **Additional information** dialog, select **.NET 7.0 (Current)** for the **Target Framework**, and then select **Create**. In **Solution Explorer**, right-click the DynamicSample project and select **Add** > **Class**. In the **Name** box, type `ReadOnlyFile`, and then select **Add**. At the top of the *ReadOnlyFile.cs* or *ReadOnlyFile.vb* file, add the following code to import the System.IO and System.Dynamic namespaces.

```
```

```
C#

using System.IO;
using System.Dynamic;
```

The custom dynamic object uses an enum to determine the search criteria. Before the class statement, add the following enum definition.

```
C#

public enum StringSearchOption
{
    StartsWith,
    Contains,
    EndsWith
}
```

Update the class statement to inherit the `DynamicObject` class, as shown in the following code example.

```
C#

class ReadOnlyFile : DynamicObject
```

Add the following code to the `ReadOnlyFile` class to define a private field for the file path and a constructor for the `ReadOnlyFile` class.

```
C#

// Store the path to the file and the initial line count value.
private string p_filePath;

// Public constructor. Verify that file exists and store the path in
// the private variable.
public ReadOnlyFile(string filePath)
{
    if (!File.Exists(filePath))
    {
        throw new Exception("File path does not exist.");
    }

    p_filePath = filePath;
}
```

1. Add the following `GetPropertyValue` method to the `ReadOnlyFile` class. The `GetPropertyValue` method takes, as input, search criteria and returns the lines from

a text file that match that search criteria. The dynamic methods provided by the `ReadOnlyFile` class call the `GetPropertyValue` method to retrieve their respective results.

C#

```
public List<string> GetPropertyValue(string propertyName,
                                      StringSearchOption StringSearchOption =
StringSearchOption.StartsWith,
                                      bool trimSpaces = true)
{
    StreamReader sr = null;
    List<string> results = new List<string>();
    string line = "";
    string testLine = "";

    try
    {
        sr = new StreamReader(p_filePath);

        while (!sr.EndOfStream)
        {
            line = sr.ReadLine();

            // Perform a case-insensitive search by using the specified
search options.
            testLine = line.ToUpper();
            if (trimSpaces) { testLine = testLine.Trim(); }

            switch (StringSearchOption)
            {
                case StringSearchOption.StartsWith:
                    if (testLine.StartsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.Contains:
                    if (testLine.Contains(propertyName.ToUpper())) {
results.Add(line); }
                    break;
                case StringSearchOption.EndsWith:
                    if (testLine.EndsWith(propertyName.ToUpper())) {
results.Add(line); }
                    break;
            }
        }
    }
    catch
    {
        // Trap any exception that occurs in reading the file and return
null.
        results = null;
    }
    finally
```

```
    {
        if (sr != null) {sr.Close();}
    }

    return results;
}
```

After the `GetPropertyValue` method, add the following code to override the TryGetMember method of the DynamicObject class. The TryGetMember method is called when a member of a dynamic class is requested and no arguments are specified. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The TryGetMember method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

C#

```
// Implement the TryGetMember method of the DynamicObject class for dynamic
member calls.
public override bool TryGetMember(GetMemberBinder binder,
                                  out object result)
{
    result = GetPropertyValue(binder.Name);
    return result == null ? false : true;
}
```

After the `TryGetMember` method, add the following code to override the TryInvokeMember method of the DynamicObject class. The TryInvokeMember method is called when a member of a dynamic class is requested with arguments. The `binder` argument contains information about the referenced member, and the `result` argument references the result returned for the specified member. The `args` argument contains an array of the arguments that are passed to the member. The TryInvokeMember method returns a Boolean value that returns `true` if the requested member exists; otherwise it returns `false`.

The custom version of the `TryInvokeMember` method expects the first argument to be a value from the `StringSearchOption` enum that you defined in a previous step. The `TryInvokeMember` method expects the second argument to be a Boolean value. If one or both arguments are valid values, they're passed to the `GetPropertyValue` method to retrieve the results.

C#

```
// Implement the TryInvokeMember method of the DynamicObject class for
// dynamic member calls that have arguments.
public override bool TryInvokeMember(InvokeMemberBinder binder,
                                     object[] args,
                                     out object result)
{
    StringSearchOption StringSearchOption = StringSearchOption.StartsWith;
    bool trimSpaces = true;

    try
    {
        if (args.Length > 0) { StringSearchOption =
(StringSearchOption)args[0]; }
    }
    catch
    {
        throw new ArgumentException("StringSearchOption argument must be a
StringSearchOption enum value.");
    }

    try
    {
        if (args.Length > 1) { trimSpaces = (bool)args[1]; }
    }
    catch
    {
        throw new ArgumentException("trimSpaces argument must be a Boolean
value.");
    }

    result = GetPropertyValue(binder.Name, StringSearchOption, trimSpaces);

    return result == null ? false : true;
}
```

Save and close the file.

Create a sample text file

In **Solution Explorer**, right-click the DynamicSample project and select **Add** > **New Item**. In the **Installed Templates** pane, select **General**, and then select the **Text File** template. Leave the default name of *TextFile1.txt* in the **Name** box, and then select **Add**. Copy the following text to the *TextFile1.txt* file.

text

```
List of customers and suppliers

Supplier: Lucerne Publishing (https://www.lucernepublishing.com/)
Customer: Preston, Chris
```

```
Customer: Hines, Patrick
Customer: Cameron, Maria
Supplier: Graphic Design Institute (https://www.graphicdesigninstitute.com/)
Supplier: Fabrikam, Inc. (https://www.fabrikam.com/)
Customer: Seubert, Roxanne
Supplier: Proseware, Inc. (http://www.proseware.com/)
Customer: Adolphi, Stephan
Customer: Koch, Paul
```

Save and close the file.

Create a sample application that uses the custom dynamic object

In **Solution Explorer**, double-click the *Program.cs* file. Add the following code to the `Main` procedure to create an instance of the `ReadOnlyFile` class for the *TextFile1.txt* file. The code uses late binding to call dynamic members and retrieve lines of text that contain the string "Customer".

C#

```
dynamic rFile = new ReadOnlyFile(@"..\..\..\TextFile1.txt");
foreach (string line in rFile.Customer)
{
    Console.WriteLine(line);
}
Console.WriteLine("----------------------------");
foreach (string line in rFile.Customer(StringSearchOption.Contains, true))
{
    Console.WriteLine(line);
}
```

Save the file and press `Ctrl+F5` to build and run the application.

Call a dynamic language library

The following walkthrough creates a project that accesses a library written in the dynamic language IronPython.

To create a custom dynamic class

In Visual Studio, select **File** > **New** > **Project**. In the **Create a new project** dialog, select C#, select **Console Application**, and then select **Next**. In the **Configure your new project** dialog, enter `DynamicIronPythonSample` for the **Project name**, and then select **Next**. In

the **Additional information** dialog, select **.NET 7.0 (Current)** for the **Target Framework**, and then select **Create**. Install the IronPython NuGet package. Edit the *Program.cs* file. At the top of the file, add the following code to import the `Microsoft.Scripting.Hosting` and `IronPython.Hosting` namespaces from the IronPython libraries and the `System.Linq` namespace.

C#

```
using System.Linq;
using Microsoft.Scripting.Hosting;
using IronPython.Hosting;
```

In the Main method, add the following code to create a new `Microsoft.Scripting.Hosting.ScriptRuntime` object to host the IronPython libraries. The `ScriptRuntime` object loads the IronPython library module random.py.

C#

```
// Set the current directory to the IronPython libraries.
System.IO.Directory.SetCurrentDirectory(
   Environment.GetFolderPath(Environment.SpecialFolder.ProgramFiles) +
   @"\IronPython 2.7\Lib");

// Create an instance of the random.py IronPython library.
Console.WriteLine("Loading random.py");
ScriptRuntime py = Python.CreateRuntime();
dynamic random = py.UseFile("random.py");
Console.WriteLine("random.py loaded.");
```

After the code to load the random.py module, add the following code to create an array of integers. The array is passed to the `shuffle` method of the random.py module, which randomly sorts the values in the array.

C#

```
// Initialize an enumerable set of integers.
int[] items = Enumerable.Range(1, 7).ToArray();

// Randomly shuffle the array of integers by using IronPython.
for (int i = 0; i < 5; i++)
{
    random.shuffle(items);
    foreach (int item in items)
    {
        Console.WriteLine(item);
    }
```

```
    Console.WriteLine("--------------------");
}
```

Save the file and press `Ctrl+F5` to build and run the application.

See also

- System.Dynamic
- System.Dynamic.DynamicObject
- Using Type dynamic
- dynamic
- Implementing Dynamic Interfaces (downloadable PDF from Microsoft TechNet)

Reduce memory allocations using new C# features

Article • 10/17/2023

> **ⓘ Important**
>
> The techniques described in this section improve performance when applied to *hot paths* in your code. *Hot paths* are those sections of your codebase that are executed often and repeatedly in normal operations. Applying these techniques to code that isn't often executed will have minimal impact. Before making any changes to improve performance, it's critical to measure a baseline. Then, analyze that baseline to determine where memory bottlenecks occur. You can learn about many cross platform tools to measure your application's performance in the section on Diagnostics and instrumentation. You can practice a profiling session in the tutorial to Measure memory usage in the Visual Studio documentation.
>
> Once you've measured memory usage and have determined that you can reduce allocations, use the techniques in this section to reduce allocations. After each successive change, measure memory usage again. Make sure each change has a positive impact on the memory usage in your application.

Performance work in .NET often means removing allocations from your code. Every block of memory you allocate must eventually be freed. Fewer allocations reduce time spent in garbage collection. It allows for more predictable execution time by removing garbage collections from specific code paths.

A common tactic to reduce allocations is to change critical data structures from `class` types to `struct` types. This change impacts the semantics of using those types. Parameters and returns are now passed by value instead of by reference. The cost of copying a value is negligible if the types are small, three words or less (considering one word being of natural size of one integer). It's measurable and can have real performance impact for larger types. To combat the effect of copying, developers can pass these types by `ref` to get back the intended semantics.

The C# `ref` features give you the ability to express the desired semantics for `struct` types without negatively impacting their overall usability. Prior to these enhancements, developers needed to resort to `unsafe` constructs with pointers and raw memory to achieve the same performance impact. The compiler generates *verifiably safe code* for the new `ref` related features. *Verifiably safe code* means the compiler detects possible

buffer overruns or accessing unallocated or freed memory. The compiler detects and prevents some errors.

Pass and return by reference

Variables in C# store *values*. In `struct` types, the value is the contents of an instance of the type. In `class` types, the value is a reference to a block of memory that stores an instance of the type. Adding the `ref` modifier means that the variable stores the *reference* to the value. In `struct` types, the reference points to the storage containing the value. In `class` types, the reference points to the storage containing the reference to the block of memory.

In C#, parameters to methods are *passed by value*, and return values are *return by value*. The *value* of the argument is passed to the method. The *value* of the return argument is the return value.

The `ref`, `in`, `ref readonly`, or `out` modifier indicates that the argument is *passed by reference*. A *reference* to the storage location is passed to the method. Adding `ref` to the method signature means the return value is *returned by reference*. A *reference* to the storage location is the return value.

You can also use *ref assignment* to have a variable refer to another variable. A typical assignment copies the *value* of the right hand side to the variable on the left hand side of the assignment. A *ref assignment* copies the memory location of the variable on the right hand side to the variable on the left hand side. The `ref` now refers to the original variable:

C#

```
int anInteger = 42; // assignment.
ref int location = ref anInteger; // ref assignment.
ref int sameLocation = ref location; // ref assignment

Console.WriteLine(location); // output: 42

sameLocation = 19; // assignment

Console.WriteLine(anInteger); // output: 19
```

When you *assign* a variable, you change its *value*. When you *ref assign* a variable, you change what it refers to.

You can work directly with the storage for values using `ref` variables, pass by reference, and ref assignment. Scope rules enforced by the compiler ensure safety when working directly with storage.

The `ref readonly` and `in` modifiers both indicate that the argument should be passed by reference and can't be reassigned in the method. The difference is that `ref readonly` indicates that the method uses the parameter as a variable. The method might capture the parameter, or it might return the parameter by readonly reference. In those cases, you should use the `ref readonly` modifier. Otherwise, the `in` modifier offers more flexibility. You don't need to add the `in` modifier to an argument for an `in` parameter, so you can update existing API signatures safely using the `in` modifier. The compiler issues a warning if you don't add either the `ref` or `in` modifier to an argument for a `ref readonly` parameter.

Ref safe context

C# includes rules for `ref` expressions to ensure that a `ref` expression can't be accessed where the storage it refers to is no longer valid. Consider the following example:

C#

```
public ref int CantEscape()
{
    int index = 42;
    return ref index; // Error: index's ref safe context is the body of
CantEscape
}
```

The compiler reports an error because you can't return a reference to a local variable from a method. The caller can't access the storage being referred to. The *ref safe context* defines the scope in which a `ref` expression is safe to access or modify. The following table lists the *ref safe contexts* for variable types. `ref` fields can't be declared in a `class` or a non-ref `struct`, so those rows aren't in the table:

Expand table

Declaration	***ref safe context***
non-ref local	block where local is declared
non-ref parameter	current method
`ref`, `ref readonly`, `in` parameter	calling method

Declaration	*ref safe context*
`out` parameter	current method
`class` field	calling method
non-ref `struct` field	current method
`ref` field of `ref struct`	calling method

A variable can be `ref` returned if its *ref safe context* is the calling method. If its *ref safe context* is the current method or a block, `ref` return is disallowed. The following snippet shows two examples. A member field can be accessed from the scope calling a method, so a class or struct field's *ref safe context* is the calling method. The *ref safe context* for a parameter with the `ref`, or `in` modifiers is the entire method. Both can be `ref` returned from a member method:

C#

```
private int anIndex;

public ref int RetrieveIndexRef()
{
    return ref anIndex;
}

public ref int RefMin(ref int left, ref int right)
{
    if (left < right)
        return ref left;
    else
        return ref right;
}
```

> **Note**
>
> When the `ref readonly` or `in` modifier is applied to a parameter, that parameter can be returned by `ref readonly`, not `ref`.

The compiler ensures that a reference can't escape its *ref safe context*. You can use `ref` parameters, `ref return`, and `ref` local variables safely because the compiler detects if you've accidentally written code where a `ref` expression could be accessed when its storage isn't valid.

Safe context and ref structs

`ref struct` types require more rules to ensure they can be used safely. A `ref struct` type can include `ref` fields. That requires the introduction of a *safe context*. For most types, the *safe context* is the calling method. In other words, a value that's not a `ref struct` can always be returned from a method.

Informally, the *safe context* for a `ref struct` is the scope where all of its `ref` fields can be accessed. In other words, it's the intersection of the *ref safe context* of all its `ref` fields. The following method returns a `ReadOnlySpan<char>` to a member field, so its *safe context* is the method:

C#

```
private string longMessage = "This is a long message";

public ReadOnlySpan<char> Safe()
{
    var span = longMessage.AsSpan();
    return span;
}
```

In contrast, the following code emits an error because the `ref field` member of the `Span<int>` refers to the stack allocated array of integers. It can't escape the method:

C#

```
public Span<int> M()
{
    int length = 3;
    Span<int> numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
    return numbers; // Error! numbers can't escape this method.
}
```

Unify memory types

The introduction of System.Span<T> and System.Memory<T> provide a unified model for working with memory. System.ReadOnlySpan<T> and System.ReadOnlyMemory<T> provide readonly versions for accessing memory. They all provide an abstraction over a block of memory storing an array of similar elements. The difference is that `Span<T>` and

`ReadOnlySpan<T>` are `ref struct` types whereas `Memory<T>` and `ReadOnlyMemory<T>` are `struct` types. Spans contain a `ref field`. Therefore instances of a span can't leave its *safe context*. The *safe context* of a `ref struct` is the *ref safe context* of its `ref field`. The implementation of `Memory<T>` and `ReadOnlyMemory<T>` remove this restriction. You use these types to directly access memory buffers.

Improve performance with ref safety

Using these features to improve performance involves these tasks:

- *Avoid allocations*: When you change a type from a `class` to a `struct`, you change how it's stored. Local variables are stored on the stack. Members are stored inline when the container object is allocated. This change means fewer allocations and that decreases the work the garbage collector does. It might also decrease memory pressure so the garbage collector runs less often.
- *Preserve reference semantics*: Changing a type from a `class` to a `struct` changes the semantics of passing a variable to a method. Code that modified the state of its parameters needs modification. Now that the parameter is a `struct`, the method is modifying a copy of the original object. You can restore the original semantics by passing that parameter as a `ref` parameter. After that change, the method modifies the original `struct` again.
- *Avoid copying data*: Copying larger `struct` types can impact performance in some code paths. You can also add the `ref` modifier to pass larger data structures to methods by reference instead of by value.
- *Restrict modifications*: When a `struct` type is passed by reference, the called method could modify the state of the struct. You can replace the `ref` modifier with the `ref readonly` or `in` modifiers to indicate that the argument can't be modified. Prefer `ref readonly` when the method captures the parameter or returns it by readonly reference. You can also create `readonly struct` types or `struct` types with `readonly` members to provide more control over what members of a `struct` can be modified.
- *Directly manipulate memory*: Some algorithms are most efficient when treating data structures as a block of memory containing a sequence of elements. The `Span` and `Memory` types provide safe access to blocks of memory.

None of these techniques require `unsafe` code. Used wisely, you can get performance characteristics from safe code that was previously only possible by using unsafe techniques. You can try the techniques yourself in the tutorial on reducing memory allocations.

Tutorial: Reduce memory allocations with `ref` safety

Article • 10/13/2023

Often, performance tuning for a .NET application involves two techniques. First, reduce the number and size of heap allocations. Second, reduce how often data is copied. Visual Studio provides great tools that help analyze how your application is using memory. Once you've determined where your app makes unnecessary allocations, you make changes to minimize those allocations. You convert `class` types to `struct` types. You use `ref` safety features to preserve semantics and minimize extra copying.

Use Visual Studio 17.5 for the best experience with this tutorial. The .NET object allocation tool used to analyze memory usage is part of Visual Studio. You can use Visual Studio Code and the command line to run the application and make all the changes. However, you won't be able to see the analysis results of your changes.

The application you'll use is a simulation of an IoT application that monitors several sensors to determine if an intruder has entered a secret gallery with valuables. The IoT sensors are constantly sending data that measures the mix of Oxygen (O2) and Carbon Dioxide (CO2) in the air. They also report the temperature and relative humidity. Each of these values is fluctuating slightly all the time. However, when a person enters the room, the change a bit more, and always in the same direction: Oxygen decreases, Carbon Dioxide increases, temperature increases, as does relative humidity. When the sensors combine to show increases, the intruder alarm is triggered.

In this tutorial, you'll run the application, take measurements on memory allocations, then improve the performance by reducing the number of allocations. The source code is available in the samples browser.

Explore the starter application

Download the application and run the starter sample. The starter application works correctly, but because it allocates many small objects with each measurement cycle, its performance slowly degrades as it runs over time.

Console

```
Press <return> to start simulation

Debounced measurements:
    Temp:      67.332
```

```
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906
Average measurements:
    Temp:      67.332
    Humidity:  41.077%
    Oxygen:    21.097%
    CO2 (ppm): 404.906

Debounced measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
Average measurements:
    Temp:      67.349
    Humidity:  46.605%
    Oxygen:    20.998%
    CO2 (ppm): 408.707
```

Many rows removed.

Console

```
Debounced measurements:
    Temp:      67.597
    Humidity:  46.543%
    Oxygen:    19.021%
    CO2 (ppm): 429.149
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High

Debounced measurements:
    Temp:      67.602
    Humidity:  46.835%
    Oxygen:    19.003%
    CO2 (ppm): 429.393
Average measurements:
    Temp:      67.568
    Humidity:  45.684%
    Oxygen:    19.631%
    CO2 (ppm): 423.498
Current intruders: 3
Calculated intruder risk: High
```

You can explore the code to learn how the application works. The main program runs the simulation. After you press `<Enter>`, it creates a room, and gathers some initial

baseline data:

```C#
Console.WriteLine("Press <return> to start simulation");
Console.ReadLine();
var room = new Room("gallery");
var r = new Random();

int counter = 0;

room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        Console.WriteLine();
        counter++;
        return counter < 20000;
    });
```

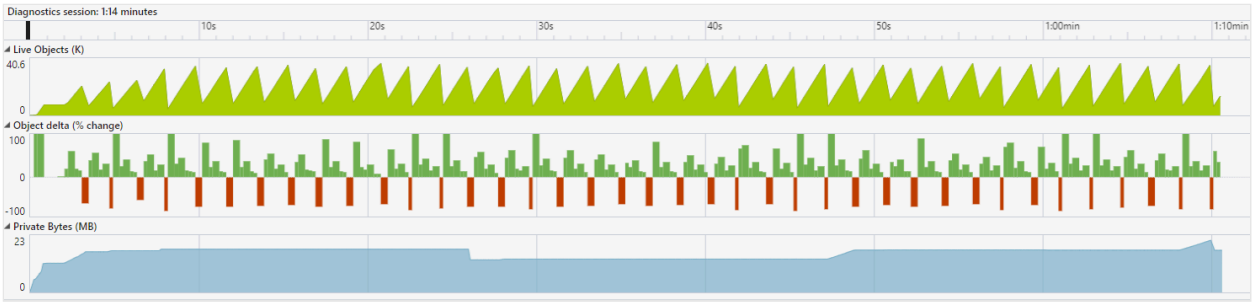
Once that baseline data has been established, it runs the simulation on the room, where a random number generator determines if an intruder has entered the room:

```C#
counter = 0;
room.TakeMeasurements(
    m =>
    {
        Console.WriteLine(room.Debounce);
        Console.WriteLine(room.Average);
        room.Intruders += (room.Intruders, r.Next(5)) switch
        {
            ( > 0, 0) => -1,
            ( < 3, 1) => 1,
            _ => 0
        };

        Console.WriteLine($"Current intruders: {room.Intruders}");
        Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
        Console.WriteLine();
        counter++;
        return counter < 200000;
    });
```

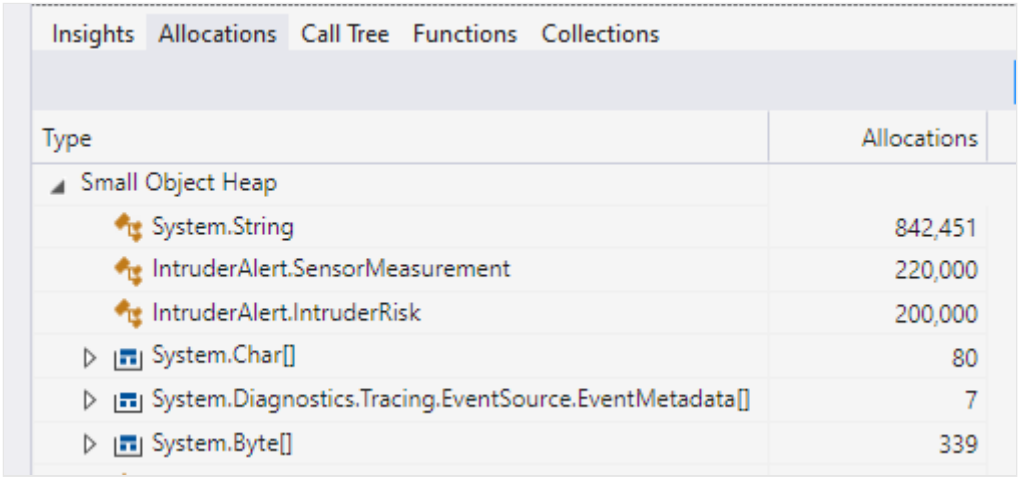
Other types contain the measurements, a debounced measurement that is the average of the last 50 measurements, and the average of all measurements taken.

Next, run the application using the .NET object allocation tool. Make sure you're using the `Release` build, not the `Debug` build. On the *Debug* menu, open the *Performance profiler*. Check the *.NET Object Allocation Tracking* option, but nothing else. Run your application to completion. The profiler measures object allocations and reports on allocations and garbage collection cycles. You should see a graph similar to the following image:



The previous graph shows that working to minimize allocations will provide performance benefits. You see a sawtooth pattern in the live objects graph. That tells you that numerous objects are created that quickly become garbage. They're later collected, as shown in the object delta graph. The downward red bars indicate a garbage collection cycle.

Next, look at the *Allocations* tab below the graphs. This table shows what types are allocated the most:



The System.String type accounts for the most allocations. The most important task should be to minimize the frequency of string allocations. This application prints numerous formatted output to the console constantly. For this simulation, we want to keep messages, so we'll concentrate on the next two rows: the `SensorMeasurement` type, and the `IntruderRisk` type.

Double-click on the `SensorMeasurement` line. You can see that all the allocations take place in the `static` method `SensorMeasurement.TakeMeasurement`. You can see the method in the following snippet:

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

Every measurement allocates a new `SensorMeasurement` object, which is a `class` type. Every `SensorMeasurement` created causes a heap allocation.

Change classes to structs

The following code shows the initial declaration of `SensorMeasurement`:

C#

```
public class SensorMeasurement
{
    private static readonly Random generator = new Random();

    public static SensorMeasurement TakeMeasurement(string room, int
intruders)
    {
        return new SensorMeasurement
        {
            CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
            O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
            Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
            Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
            Room = room,
            TimeRecorded = DateTime.Now
        };
    }
```

```
    private const double CO2Concentration = 409.8; // increases with people.
    private const double O2Concentration = 0.2100; // decreases
    private const double TemperatureSetting = 67.5; // increases
    private const double HumiditySetting = 0.4500; // increases

    public required double CO2 { get; init; }
    public required double O2 { get; init; }
    public required double Temperature { get; init; }
    public required double Humidity { get; init; }
    public required string Room { get; init; }
    public required DateTime TimeRecorded { get; init; }

    public override string ToString() => $"""
            Room: {Room} at {TimeRecorded}:
                Temp:      {Temperature:F3}
                Humidity:  {Humidity:P3}
                Oxygen:    {O2:P3}
                CO2 (ppm): {CO2:F3}
            """;
}
```

The type was originally created as a `class` because it contains numerous `double` measurements. It's larger than you'd want to copy in hot paths. However, that decision meant a large number of allocations. Change the type from a `class` to a `struct`.

Changing from a `class` to `struct` introduces a few compiler errors because the original code used `null` reference checks in a few spots. The first is in the `DebounceMeasurement` class, in the `AddMeasurement` method:

C#

```
public void AddMeasurement(SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
    {
        if (recentMeasurements[i] is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
```

```
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
}
```

The `DebounceMeasurement` type contains an array of 50 measurements. The readings for a sensor are reported as the average of the last 50 measurements. That reduces the noise in the readings. Before a full 50 readings have been taken, these values are `null`. The code checks for `null` reference to report the correct average on system startup. After changing the `SensorMeasurement` type to a struct, you must use a different test. The `SensorMeasurement` type includes a `string` for the room identifier, so you can use that test instead:

C#

```
if (recentMeasurements[i].Room is not null)
```

The other three compiler errors are all in the method that repeatedly takes measurements in a room:

C#

```
public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
{
    SensorMeasurement? measure = default;
    do {
        measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
        Average.AddMeasurement(measure);
        Debounce.AddMeasurement(measure);
    } while (MeasurementHandler(measure));
}
```

In the starter method, the local variable for the `SensorMeasurement` is a *nullable reference*:

C#

```
SensorMeasurement? measure = default;
```

Now that the `SensorMeasurement` is a `struct` instead of a `class`, the nullable is a *nullable value type*. You can change the declaration to a value type to fix the remaining compiler

errors:

```
C#

SensorMeasurement measure = default;
```

Now that the compiler errors have been addressed, you should examine the code to ensure the semantics haven't changed. Because `struct` types are passed by value, modifications made to method parameters aren't visible after the method returns.

> **ⓘ Important**
>
> Changing a type from a `class` to a `struct` can change the semantics of your program. When a `class` type is passed to a method, any mutations made in the method are made to the argument. When a `struct` type is passed to a method, and mutations made in the method are made to *a copy* of the argument. That means any method that modifies its arguments by design should be updated to use the `ref` modifier on any argument type you've changed from a `class` to a `struct`.

The `SensorMeasurement` type doesn't include any methods that change state, so that's not a concern in this sample. You can prove that by adding the `readonly` modifier to the `SensorMeasurement` struct:

```
C#

public readonly struct SensorMeasurement
```

The compiler enforces the `readonly` nature of the `SensorMeasurement` struct. If your inspection of the code missed some method that modified state, the compiler would tell you. Your app still builds without errors, so this type is `readonly`. Adding the `readonly` modifier when you change a type from a `class` to a `struct` can help you find members that modify the state of the `struct`.

Avoid making copies

You've removed a large number of unnecessary allocations from your app. The `SensorMeasurement` type doesn't appear in the table anywhere.

Now, it's doing extra working copying the `SensorMeasurement` structure every time it's used as a parameter or a return value. The `SensorMeasurement` struct contains four doubles, a DateTime and a `string`. That structure is measurably larger than a reference. Let's add the `ref` or `in` modifiers to places where the `SensorMeasurement` type is used.

The next step is to find methods that return a measurement, or take a measurement as an argument, and use references where possible. Start in the `SensorMeasurement` struct. The static `TakeMeasurement` method creates and returns a new `SensorMeasurement`:

C#

```
public static SensorMeasurement TakeMeasurement(string room, int intruders)
{
    return new SensorMeasurement
    {
        CO2 = (CO2Concentration + intruders * 10) + (20 *
generator.NextDouble() - 10.0),
        O2 = (O2Concentration - intruders * 0.01) + (0.005 *
generator.NextDouble() - 0.0025),
        Temperature = (TemperatureSetting + intruders * 0.05) + (0.5 *
generator.NextDouble() - 0.25),
        Humidity = (HumiditySetting + intruders * 0.005) + (0.20 *
generator.NextDouble() - 0.10),
        Room = room,
        TimeRecorded = DateTime.Now
    };
}
```

We'll leave this one as is, returning by value. If you tried to return by `ref`, you'd get a compiler error. You can't return a `ref` to a new structure locally created in the method. The design of the immutable struct means you can only set the values of the measurement at construction. This method must create a new measurement struct.

Let's look again at `DebounceMeasurement.AddMeasurement`. You should add the `in` modifier to the `measurement` parameter:

C#

```
public void AddMeasurement(in SensorMeasurement datum)
{
    int index = totalMeasurements % debounceSize;
    recentMeasurements[index] = datum;
    totalMeasurements++;
    double sumCO2 = 0;
    double sumO2 = 0;
    double sumTemp = 0;
    double sumHumidity = 0;
    for (int i = 0; i < debounceSize; i++)
```

```
    {
        if (recentMeasurements[i].Room is not null)
        {
            sumCO2 += recentMeasurements[i].CO2;
            sumO2+= recentMeasurements[i].O2;
            sumTemp+= recentMeasurements[i].Temperature;
            sumHumidity += recentMeasurements[i].Humidity;
        }
    }
    O2 = sumO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    CO2 = sumCO2 / ((totalMeasurements > debounceSize) ? debounceSize :
totalMeasurements);
    Temperature = sumTemp / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
    Humidity = sumHumidity / ((totalMeasurements > debounceSize) ?
debounceSize : totalMeasurements);
}
```

That saves one copy operation. The `in` parameter is a reference to the copy already created by the caller. You can also save a copy with the `TakeMeasurement` method in the `Room` type. This method illustrates how the compiler provides safety when you pass arguments by `ref`. The initial `TakeMeasurement` method in the `Room` type takes an argument of `Func<SensorMeasurement, bool>`. If you try to add the `in` or `ref` modifier to that declaration, the compiler reports an error. You can't pass a `ref` argument to a lambda expression. The compiler can't guarantee that the called expression doesn't copy the reference. If the lambda expression *captures* the reference, the reference could have a lifetime longer than the value it refers to. Accessing it outside its *ref safe context* would result in memory corruption. The `ref` safety rules don't allow it. You can learn more in the overview of ref safety features.

Preserve semantics

The final sets of changes won't have a major impact on this application's performance because the types aren't created in hot paths. These changes illustrate some of the other techniques you'd use in your performance tuning. Let's take a look at the initial `Room` class:

C#

```
public class Room
{
    public AverageMeasurement Average { get; } = new ();
    public DebounceMeasurement Debounce { get; } = new ();
    public string Name { get; }
```

```
    public IntruderRisk RiskStatus
    {
        get
        {
            var CO2Variance = (Debounce.CO2 - Average.CO2) > 10.0 / 4;
            var O2Variance = (Average.O2 - Debounce.O2) > 0.005 / 4.0;
            var TempVariance = (Debounce.Temperature - Average.Temperature)
> 0.05 / 4.0;
            var HumidityVariance = (Debounce.Humidity - Average.Humidity) >
0.20 / 4;
            IntruderRisk risk = IntruderRisk.None;
            if (CO2Variance) { risk++; }
            if (O2Variance) { risk++; }
            if (TempVariance) { risk++; }
            if (HumidityVariance) { risk++; }
            return risk;
        }
    }

    public int Intruders { get; set; }


    public Room(string name)
    {
        Name = name;
    }

    public void TakeMeasurements(Func<SensorMeasurement, bool>
MeasurementHandler)
    {
        SensorMeasurement? measure = default;
        do {
            measure = SensorMeasurement.TakeMeasurement(Name, Intruders);
            Average.AddMeasurement(measure);
            Debounce.AddMeasurement(measure);
        } while (MeasurementHandler(measure));
    }
}
```

This type contains several properties. Some are `class` types. Creating a `Room` object involves multiple allocations. One for the `Room` itself, and one for each of the members of a `class` type that it contains. You can convert two of these properties from `class` types to `struct` types: the `DebounceMeasurement` and `AverageMeasurement` types. Let's work through that transformation with both types.

Change the `DebounceMeasurement` type from a `class` to `struct`. That introduces a compiler error `CS8983: A 'struct' with field initializers must include an explicitly declared constructor`. You can fix this by adding an empty parameterless constructor:

```
C#
```

```
public DebounceMeasurement() { }
```

You can learn more about this requirement in the language reference article on structs.

The Object.ToString() override doesn't modify any of the values of the struct. You can add the `readonly` modifier to that method declaration. The `DebounceMeasurement` type is *mutable*, so you'll need to take care that modifications don't affect copies that are discarded. The `AddMeasurement` method does modify the state of the object. It's called from the `Room` class, in the `TakeMeasurements` method. You want those changes to persist after calling the method. You can change the `Room.Debounce` property to return a *reference* to a single instance of the `DebounceMeasurement` type:

C#

```
private DebounceMeasurement debounce = new();
public ref readonly DebounceMeasurement Debounce { get { return ref
debounce; } }
```

There are a few changes in the previous example. First, the *property* is a readonly property that returns a readonly reference to the instance owned by this room. It's now backed by a declared field that's initialized when the `Room` object is instantiated. After making these changes, you'll update the implementation of `AddMeasurement` method. It uses the private backing field, `debounce`, not the readonly property `Debounce`. That way, the changes take place on the single instance created during initialization.

The same technique works with the `Average` property. First, you modify the `AverageMeasurement` type from a `class` to a `struct`, and add the `readonly` modifier on the `ToString` method:

C#

```
namespace IntruderAlert;

public struct AverageMeasurement
{
    private double sumCO2 = 0;
    private double sumO2 = 0;
    private double sumTemperature = 0;
    private double sumHumidity = 0;
    private int totalMeasurements = 0;

    public AverageMeasurement() { }

    public readonly double CO2 => sumCO2 / totalMeasurements;
    public readonly double O2 => sumO2 / totalMeasurements;
```

```
    public readonly double Temperature => sumTemperature /
totalMeasurements;
    public readonly double Humidity => sumHumidity / totalMeasurements;

    public void AddMeasurement(in SensorMeasurement datum)
    {
        totalMeasurements++;
        sumCO2 += datum.CO2;
        sumO2 += datum.O2;
        sumTemperature += datum.Temperature;
        sumHumidity+= datum.Humidity;
    }

    public readonly override string ToString() => $"""
        Average measurements:
            Temp:      {Temperature:F3}
            Humidity:  {Humidity:P3}
            Oxygen:    {O2:P3}
            CO2 (ppm): {CO2:F3}
        """;
}
```

Then, you modify the `Room` class following the same technique you used for the `Debounce` property. The `Average` property returns a `readonly ref` to the private field for the average measurement. The `AddMeasurement` method modifies the internal fields.

C#

```
private AverageMeasurement average = new();
public  ref readonly AverageMeasurement Average { get { return ref average;
} }
```

Avoid boxing

There's one final change to improve performance. The main program is printing stats for the room, including the risk assessment:

C#

```
Console.WriteLine($"Current intruders: {room.Intruders}");
Console.WriteLine($"Calculated intruder risk: {room.RiskStatus}");
```

The call to the generated `ToString` boxes the enum value. You can avoid that by writing an override in the `Room` class that formats the string based on the value of estimated risk:

C#

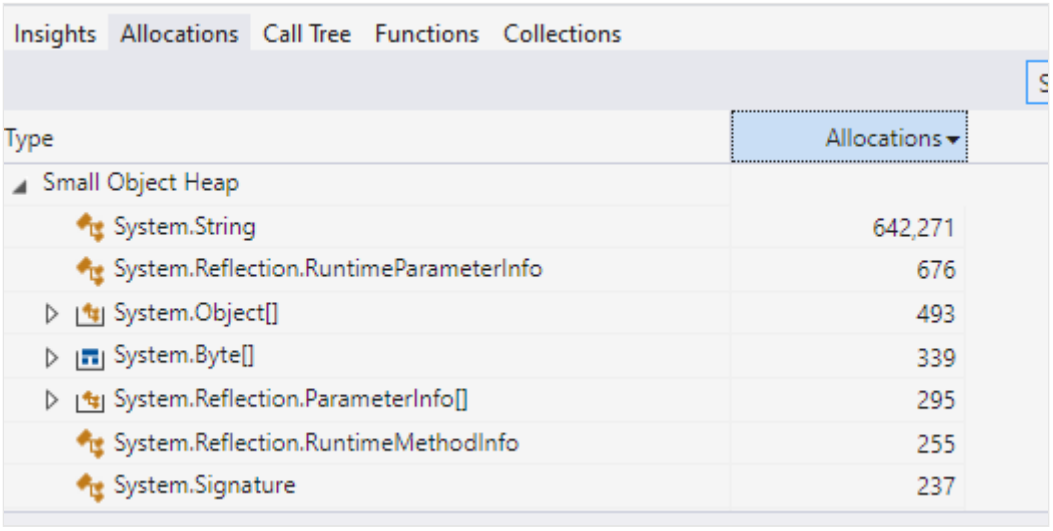
```
public override string ToString() =>
    $"Calculated intruder risk: {RiskStatus switch
    {
        IntruderRisk.None => "None",
        IntruderRisk.Low => "Low",
        IntruderRisk.Medium => "Medium",
        IntruderRisk.High => "High",
        IntruderRisk.Extreme => "Extreme",
        _ => "Error!"
    }}, Current intruders: {Intruders.ToString()}";
```

Then, modify the code in the main program to call this new `ToString` method:

C#

```
Console.WriteLine(room.ToString());
```

Run the app using the profiler and look at the updated table for allocations.



You've removed numerous allocations, and provided your app with a performance boost.

Using ref safety in your application

These techniques are low-level performance tuning. They can increase performance in your application when applied to hot paths, and when you've measured the impact before and after the changes. In most cases, the cycle you'll follow is:

- *Measure allocations*: Determine what types are being allocated the most, and when you can reduce the heap allocations.

- *Convert class to struct*: Many times, types can be converted from a `class` to a `struct`. Your app uses stack space instead of making heap allocations.
- *Preserve semantics*: Converting a `class` to a `struct` can impact the semantics for parameters and return values. Any method that modifies its parameters should now mark those parameters with the `ref` modifier. That ensures the modifications are made to the correct object. Similarly, if a property or method return value should be modified by the caller, that return should be marked with the `ref` modifier.
- *Avoid copies*: When you pass a large struct as a parameter, you can mark the parameter with the `in` modifier. You can pass a reference in fewer bytes, and ensure that the method doesn't modify the original value. You can also return values by `readonly ref` to return a reference that can't be modified.

Using these techniques you can improve performance in hot paths of your code.

Tutorial: Write a custom string interpolation handler

Summarize this article for me

In this tutorial, you learn how to:

- ✓ Implement the string interpolation handler pattern.
- ✓ Interact with the receiver in a string interpolation operation.
- ✓ Add arguments to the string interpolation handler.
- ✓ Understand the new library features for string interpolation.

Prerequisites

Set up your machine to run .NET. The C# compiler is available through Visual Studio or the .NET SDK.

This tutorial assumes you're familiar with C# and .NET, including either Visual Studio or Visual Studio Code and the C# DevKit.

You can write a custom *interpolated string handler*. An interpolated string handler is a type that processes the placeholder expression in an interpolated string. Without a custom handler, the system processes placeholders similar to String.Format. Each placeholder is formatted as text, and then the components are concatenated to form the resulting string.

You can write a handler for any scenario where you use information about the resulting string. Consider questions like: Is it used? What constraints are on the format? Some examples include:

- You might require none of the resulting strings are greater than some limit, such as 80 characters. You can process the interpolated strings to fill a fixed-length buffer, and stop processing once that buffer length is reached.
- You might have a tabular format, and each placeholder must have a fixed length. A custom handler can enforce that constraint, rather than forcing all client code to conform.

In this tutorial, you create a string interpolation handler for one of the core performance scenarios: logging libraries. Depending on the configured log level, the work to construct a log message isn't needed. If logging is off, the work to construct a string from an interpolated string expression isn't needed. The message is never printed, so any string concatenation can be skipped. In addition, any expressions used in the placeholders, including generating stack traces, don't need to be done.

An interpolated string handler can determine if the formatted string is used, and only perform the necessary work if needed.

Initial implementation

Start with a basic `Logger` class that supports different levels:

C#

```
public enum LogLevel
{
    Off,
    Critical,
    Error,
    Warning,
    Information,
    Trace
}

public class Logger
{
    public LogLevel EnabledLevel { get; init; } = LogLevel.Error;

    public void LogMessage(LogLevel level, string msg)
    {
        if (EnabledLevel < level) return;
        Console.WriteLine(msg);
    }
}
```

This `Logger` supports six different levels. When a message doesn't pass the log level filter, the logger produces no output. The public API for the logger accepts a fully formatted string as the message. The caller does all the work to create the string.

Implement the handler pattern

In this step, you build an *interpolated string handler* that recreates the current behavior. An interpolated string handler is a type that must have the following characteristics:

- The System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute applied to the type.
- A constructor that has two `int` parameters, `literalLength` and `formattedCount`. (More parameters are allowed).
- A public `AppendLiteral` method with the signature: `public void AppendLiteral(string s)`.

- A generic public `AppendFormatted` method with the signature: `public void AppendFormatted<T>(T t)`.

Internally, the builder creates the formatted string and provides a member for a client to retrieve that string. The following code shows a `LogInterpolatedStringHandler` type that meets these requirements:

C#

```
[InterpolatedStringHandler]
public struct LogInterpolatedStringHandler
{
    // Storage for the built-up string
    StringBuilder builder;

    public LogInterpolatedStringHandler(int literalLength, int formattedCount)
    {
        builder = new StringBuilder(literalLength);
        Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    }

    public void AppendLiteral(string s)
    {
        Console.WriteLine($"\tAppendLiteral called: {{{s}}}");

        builder.Append(s);
        Console.WriteLine($"\tAppended the literal string");
    }

    public void AppendFormatted<T>(T t)
    {
        Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type
{typeof(T)}");

        builder.Append(t?.ToString());
        Console.WriteLine($"\tAppended the formatted object");
    }

    public override string ToString() => builder.ToString();
}
```

> **Note**
>
> When the interpolated string expression is a compile-time constant (that is, it has no placeholders), the compiler uses the target type `string` instead of invoking a custom interpolated string handler. This behavior means constant interpolated strings bypass custom handlers entirely.

You can now add an overload to `LogMessage` in the `Logger` class to try your new interpolated string handler:

C#

```
public void LogMessage(LogLevel level, LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.ToString());
}
```

You don't need to remove the original `LogMessage` method. When the argument is an interpolated string expression, the compiler prefers a method with an interpolated handler parameter over a method with a `string` parameter.

You can verify that the new handler is invoked by using the following code as the main program:

C#

```
var logger = new Logger() { EnabledLevel = LogLevel.Warning };
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. This is an
error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time}. This won't be
printed.");
logger.LogMessage(LogLevel.Warning, "Warning Level. This warning is a string, not
an interpolated string expression.");
```

Running the application produces output similar to the following text:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be
printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This won't be printed.}
```

```
        Appended the literal string
Warning Level. This warning is a string, not an interpolated string expression.
```

Tracing through the output, you can see how the compiler adds code to call the handler and build the string:

- The compiler adds a call to construct the handler, passing the total length of the literal text in the format string, and the number of placeholders.
- The compiler adds calls to `AppendLiteral` and `AppendFormatted` for each section of the literal string and for each placeholder.
- The compiler invokes the `LogMessage` method using the `CoreInterpolatedStringHandler` as the argument.

Finally, notice that the last warning doesn't invoke the interpolated string handler. The argument is a `string`, so that call invokes the other overload with a string parameter.

> **ⓘ Important**
>
> Use `ref struct` for interpolated string handlers only if absolutely necessary. `ref struct` types have limitations as they must be stored on the stack. For example, they don't work if an interpolated string hole contains an `await` expression because the compiler needs to store the handler in the compiler-generated `IAsyncStateMachine` implementation.

Add more capabilities to the handler

The preceding version of the interpolated string handler implements the pattern. To avoid processing every placeholder expression, you need more information in the handler. In this section, you improve your handler so that it does less work when the constructed string isn't written to the log. You use System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute to specify a mapping between parameters to a public API and parameters to a handler's constructor. That mapping provides the handler with the information needed to determine if the interpolated string should be evaluated.

Start with changes to the handler. First, add a field to track if the handler is enabled. Add two parameters to the constructor: one to specify the log level for this message, and the other a reference to the log object:

C#

```
private readonly bool enabled;
```

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount, Logger
logger, LogLevel logLevel)
{
    enabled = logger.EnabledLevel >= logLevel;
    builder = new StringBuilder(literalLength);
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
}
```

Next, use the field so that your handler only appends literals or formatted objects when the final string is used:

C#

```
public void AppendLiteral(string s)
{
    Console.WriteLine($"\tAppendLiteral called: {{{s}}}");
    if (!enabled) return;

    builder.Append(s);
    Console.WriteLine($"\tAppended the literal string");
}

public void AppendFormatted<T>(T t)
{
    Console.WriteLine($"\tAppendFormatted called: {{{t}}} is of type {typeof(T)}");
    if (!enabled) return;

    builder.Append(t?.ToString());
    Console.WriteLine($"\tAppended the formatted object");
}
```

Next, update the `LogMessage` declaration so that the compiler passes the additional parameters to the handler's constructor. Handle this step by using the System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute on the handler argument:

C#

```
public void LogMessage(LogLevel level, [InterpolatedStringHandlerArgument("",
"level")] LogInterpolatedStringHandler builder)
{
    if (EnabledLevel < level) return;
    Console.WriteLine(builder.ToString());
}
```

This attribute specifies the list of arguments to `LogMessage` that map to the parameters that follow the required `literalLength` and `formattedCount` parameters. The empty string (`""`), specifies the receiver. The compiler substitutes the value of the `Logger` object represented by

`this` for the next argument to the handler's constructor. The compiler substitutes the value of `level` for the following argument. You can provide any number of arguments for any handler you write. The arguments that you add are string arguments.

> **Note**
>
> If the InterpolatedStringHandlerArgumentAttribute constructor argument list is empty, the behavior is the same as if the attribute was omitted entirely.

You can run this version by using the same test code. This time, you see the following results:

PowerShell

```
        literal length: 65, formattedCount: 1
        AppendLiteral called: {Error Level. CurrentTime: }
        Appended the literal string
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: {. This is an error. It will be printed.}
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. This is an error. It will be
printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: {Trace Level. CurrentTime: }
        AppendFormatted called: {10/20/2021 12:19:10 PM} is of type System.DateTime
        AppendLiteral called: {. This won't be printed.}
Warning Level. This warning is a string, not an interpolated string expression.
```

You see that the `AppendLiteral` and `AppendFormat` methods are called, but they aren't doing any work. The handler determined that the final string isn't needed, so the handler doesn't build it. There are still a couple of improvements to make.

First, you can add an overload of `AppendFormatted` that constrains the argument to a type that implements System.IFormattable. This overload enables callers to add format strings in the placeholders. While making this change, also change the return type of the other `AppendFormatted` and `AppendLiteral` methods, from `void` to `bool`. If any of these methods have different return types, you get a compilation error. That change enables *short circuiting*. The methods return `false` to indicate that processing of the interpolated string expression should be stopped. Returning `true` indicates that it should continue. In this example, you're using it to stop processing when the resulting string isn't needed. Short circuiting supports more fine-grained actions. You could stop processing the expression once it reaches a certain length, to support fixed-length buffers. Or some condition could indicate remaining elements aren't needed.

C#

```
public void AppendFormatted<T>(T t, string format) where T : IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t} with
format {{{format}}} is of type {typeof(T)},");

    builder.Append(t?.ToString(format, null));
    Console.WriteLine($"\tAppended the formatted object");
}

public void AppendFormatted<T>(T t, int alignment, string format) where T :
IFormattable
{
    Console.WriteLine($"\tAppendFormatted (IFormattable version) called: {t} with
alignment {alignment} and format {{{format}}} is of type {typeof(T)},");
    var formatString =$"{alignment}:{format}";
    builder.Append(string.Format($"{{0,{formatString}}}", t));
    Console.WriteLine($"\tAppended the formatted object");
}
```

With that addition, you can specify format strings in your interpolated string expression:

C#

```
var time = DateTime.Now;

logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time}. The time
doesn't use formatting.");
logger.LogMessage(LogLevel.Error, $"Error Level. CurrentTime: {time:t}. This is an
error. It will be printed.");
logger.LogMessage(LogLevel.Trace, $"Trace Level. CurrentTime: {time:t}. This won't
be printed.");
```

The `:t` on the first message specifies the "short time format" for the current time. The previous example showed one of the overloads to the `AppendFormatted` method that you can create for your handler. You don't need to specify a generic argument for the object being formatted. You might have more efficient ways to convert types you create to string. You can write overloads of `AppendFormatted` that takes those types instead of a generic argument. The compiler picks the best overload. The runtime uses this technique to convert System.Span<T> to string output. You can add an integer parameter to specify the *alignment* of the output, with or without an IFormattable. The System.Runtime.CompilerServices.DefaultInterpolatedStringHandler that ships with .NET 6 contains nine overloads of AppendFormatted for different uses. You can use it as a reference while building a handler for your purposes.

Run the sample now, and you see that for the `Trace` message, only the first `AppendLiteral` is called:

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:18:29 PM is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:18:29 PM. The time doesn't use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:18:29 PM with
format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:18 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
        AppendLiteral called: Trace Level. CurrentTime:
Warning Level. This warning is a string, not an interpolated string expression.
```

You can make one final update to the handler's constructor that improves efficiency. The handler can add a final `out bool` parameter. Setting that parameter to `false` indicates that the handler shouldn't be called at all to process the interpolated string expression:

C#

```
public LogInterpolatedStringHandler(int literalLength, int formattedCount, Logger
logger, LogLevel level, out bool isEnabled)
{
    isEnabled = logger.EnabledLevel >= level;
    Console.WriteLine($"\tliteral length: {literalLength}, formattedCount:
{formattedCount}");
    builder = isEnabled ? new StringBuilder(literalLength) : default!;
}
```

That change means you can remove the `enabled` field. Then, you can change the return type of `AppendLiteral` and `AppendFormatted` to `void`. Now, when you run the sample, you see the following output:

PowerShell

```
        literal length: 60, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted called: 10/20/2021 12:19:10 PM is of type System.DateTime
        Appended the formatted object
        AppendLiteral called: . The time doesn't use formatting.
```

```
        Appended the literal string
Error Level. CurrentTime: 10/20/2021 12:19:10 PM. The time doesn't use formatting.
        literal length: 65, formattedCount: 1
        AppendLiteral called: Error Level. CurrentTime:
        Appended the literal string
        AppendFormatted (IFormattable version) called: 10/20/2021 12:19:10 PM with
format {t} is of type System.DateTime,
        Appended the formatted object
        AppendLiteral called: . This is an error. It will be printed.
        Appended the literal string
Error Level. CurrentTime: 12:19 PM. This is an error. It will be printed.
        literal length: 50, formattedCount: 1
Warning Level. This warning is a string, not an interpolated string expression.
```

The only output when `LogLevel.Trace` was specified is the output from the constructor. The handler indicated that it's not enabled, so none of the `Append` methods are invoked.

This example illustrates an important point for interpolated string handlers, especially when logging libraries are used. Any side-effects in the placeholders might not occur. Add the following code to your main program and see this behavior in action:

C#

```
int index = 0;
int numberOfIncrements = 0;
for (var level = LogLevel.Critical; level <= LogLevel.Trace; level++)
{
    Console.WriteLine(level);
    logger.LogMessage(level, $"{level}: Increment index {index++}");
    numberOfIncrements++;
}
Console.WriteLine($"Value of index {index}, value of numberOfIncrements:
{numberOfIncrements}");
```

You can see the `index` variable is incremented each iteration of the loop. Because the placeholders are evaluated only for `Critical`, `Error`, and `Warning` levels, not for `Information`, and `Trace`, the final value of `index` doesn't match the expectation:

PowerShell

```
Critical
Critical: Increment index 0
Error
Error: Increment index 1
Warning
Warning: Increment index 2
Information
Trace
Value of index 3, value of numberOfIncrements: 5
```

Interpolated string handlers provide greater control over how an interpolated string expression is converted to a string. The .NET runtime team used this feature to improve performance in several areas. You can make use of the same capability in your own libraries. To explore further, look at the System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. It provides a more complete implementation than you built here. You see many more overloads that are possible for the `Append` methods.

Last updated on 02/05/2026

The .NET Compiler Platform SDK

Article • 10/25/2024

Compilers build a detailed model of application code as they validate the syntax and semantics of that code. They use this model to build the executable output from the source code. The .NET Compiler Platform SDK provides access to this model. Increasingly, we rely on integrated development environment (IDE) features such as IntelliSense, refactoring, intelligent rename, "Find all references," and "Go to definition" to increase our productivity. We rely on code analysis tools to improve our code quality, and code generators to aid in application construction. As these tools get smarter, they need access to more and more of the model that only compilers create as they process application code. This is the core mission of the Roslyn APIs: opening up the opaque boxes and allowing tools and end users to share in the wealth of information compilers have about our code. Instead of being opaque source-code-in and object-code-out translators, through Roslyn, compilers become platforms: APIs that you can use for code-related tasks in your tools and applications.

.NET Compiler Platform SDK concepts

The .NET Compiler Platform SDK dramatically lowers the barrier to entry for creating code focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and Visual Basic languages, and embedding of C# and Visual Basic in domain-specific languages.

The .NET Compiler Platform SDK enables you to build ***analyzers*** and ***code fixes*** that find and correct coding mistakes. ***Analyzers*** understand the syntax (structure of code) and semantics to detect practices that should be corrected. ***Code fixes*** provide one or more suggested fixes for addressing coding mistakes found by analyzers or compiler diagnostics. Typically, an analyzer and the associated code fixes are packaged together in a single project.

Analyzers and code fixes use static analysis to understand code. They do not run the code or provide other testing benefits. They can, however, point out practices that often lead to bugs, unmaintainable code, or standard guideline violation.

In addition to analyzers and code fixes, The .NET Compiler Platform SDK also enables you to build ***code refactorings***. It also provides a single set of APIs that enable you to examine and understand a C# or Visual Basic codebase. Because you can use this single codebase, you can write analyzers and code fixes more easily by leveraging the syntactic

and semantic analysis APIs provided by the .NET Compiler Platform SDK. Freed from the large task of replicating the analysis done by the compiler, you can concentrate on the more focused task of finding and fixing common coding errors for your project or library.

A smaller benefit is that your analyzers and code fixes are smaller and use much less memory when loaded in Visual Studio than they would if you wrote your own codebase to understand the code in a project. By leveraging the same classes used by the compiler and Visual Studio, you can create your own static analysis tools. This means your team can use analyzers and code fixes without a noticeable impact on the IDE's performance.

There are three main scenarios for writing analyzers and code fixes:

1. *Enforce team coding standards*
2. *Provide guidance with library packages*
3. *Provide general guidance*

Enforce team coding standards

Many teams have coding standards that are enforced through code reviews with other team members. Analyzers and code fixes can make this process much more efficient. Code reviews happen when a developer shares their work with others on the team. The developer will have invested all the time needed to complete a new feature before getting any comments. Weeks may go by while the developer reinforces habits that don't match the team's practices.

Analyzers run as a developer writes code. The developer gets immediate feedback that encourages following the guidance immediately. The developer builds habits to write compliant code as soon as they begin prototyping. When the feature is ready for humans to review, all the standard guidance has been enforced.

Teams can build analyzers and code fixes that look for the most common practices that violate team coding practices. These can be installed on each developer's machine to enforce the standards.

> **Tip**
>
> Before building your own analyzer, check out the built-in ones. For more information, see **Code-style rules**.

Provide guidance with library packages

There is a wealth of libraries available for .NET developers on NuGet. Some of these come from Microsoft, some from third-party companies, and others from community members and volunteers. These libraries get more adoption and higher reviews when developers can succeed with those libraries.

In addition to providing documentation, you can provide analyzers and code fixes that find and correct common mis-uses of your library. These immediate corrections will help developers succeed more quickly.

You can package analyzers and code fixes with your library on NuGet. In that scenario, every developer who installs your NuGet package will also install the analyzer package. All developers using your library will immediately get guidance from your team in the form of immediate feedback on mistakes and suggested corrections.

Provide general guidance

The .NET developer community has discovered, through experience, patterns that work well and patterns that are best avoided. Several community members have created analyzers that enforce those recommended patterns. As we learn more, there is always room for new ideas.

These analyzers can be uploaded to the Visual Studio Marketplace and downloaded by developers using Visual Studio. Newcomers to the language and the platform learn accepted practices quickly and become productive earlier in their .NET journey. As these become more widely used, the community adopts these practices.

Source generators

Source generators aim to enable *compile time metaprogramming*, that is, code that can be created at compile time and added to the compilation. Source generators are able to read the contents of the compilation before running, as well as access any *additional files*. This ability enables them to introspect both user C# code and generator-specific files. You can learn how to build incremental source generators using the source generator cookbook.

Next steps

The .NET Compiler Platform SDK includes the latest language object models for code generation, analysis, and refactoring. This section provides a conceptual overview of the

.NET Compiler Platform SDK. Further details can be found in the quickstarts, samples, and tutorials sections.

You can learn more about the concepts in the .NET Compiler Platform SDK in these five topics:

- Explore code with the syntax visualizer
- Understand the compiler API model
- Work with syntax
- Work with semantics
- Work with a workspace

To get started, you'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab

4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

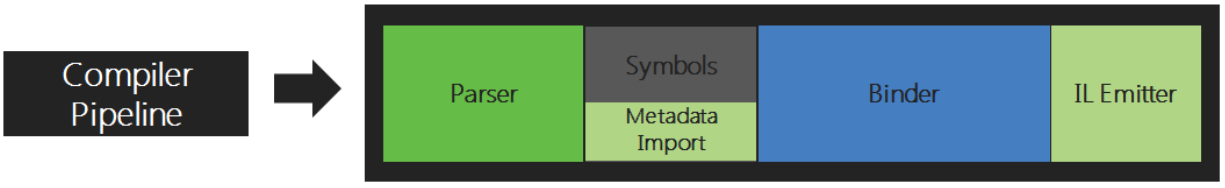
Understand the .NET Compiler Platform SDK model

Article • 09/15/2021

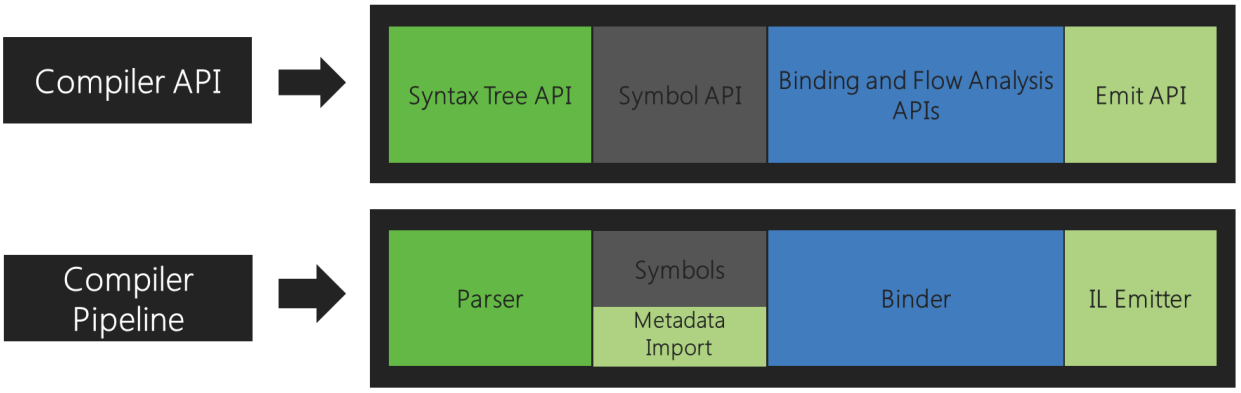
Compilers process the code you write following structured rules that often differ from the way humans read and understand code. A basic understanding of the model used by compilers is essential to understanding the APIs you use when building Roslyn-based tools.

Compiler pipeline functional areas

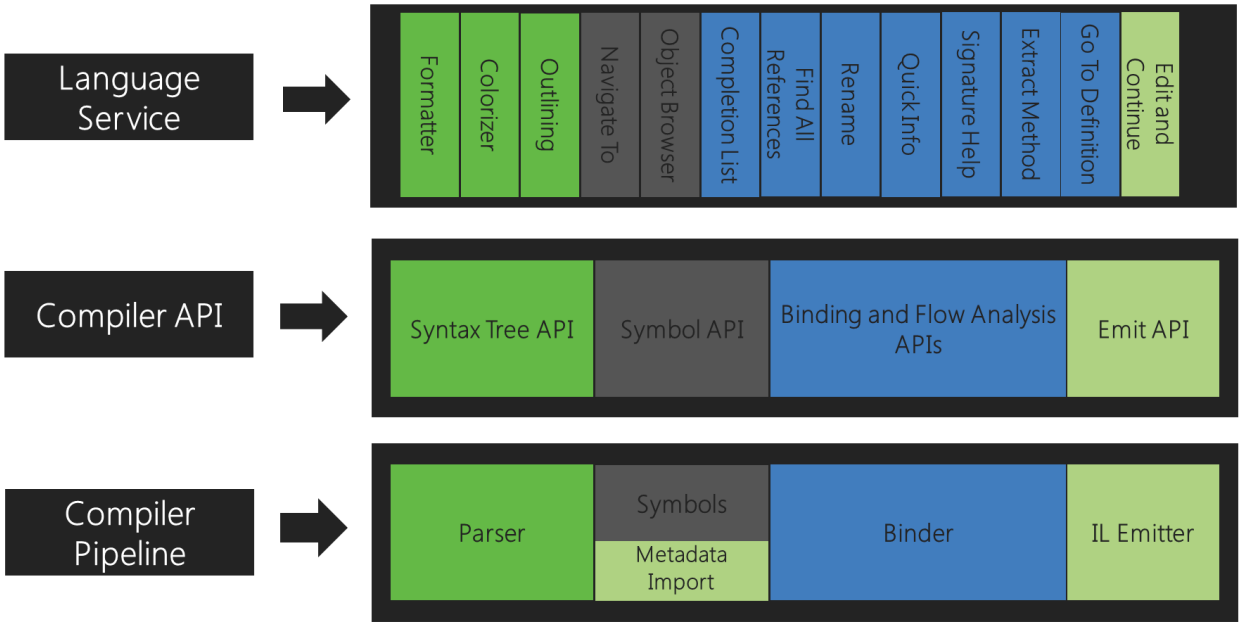
The .NET Compiler Platform SDK exposes the C# and Visual Basic compilers' code analysis to you as a consumer by providing an API layer that mirrors a traditional compiler pipeline.



Each phase of this pipeline is a separate component. First, the parse phase tokenizes and parses source text into syntax that follows the language grammar. Second, the declaration phase analyzes source and imported metadata to form named symbols. Next, the bind phase matches identifiers in the code to symbols. Finally, the emit phase emits an assembly with all the information built up by the compiler.



Corresponding to each of those phases, the .NET Compiler Platform SDK exposes an object model that allows access to the information at that phase. The parsing phase exposes a syntax tree, the declaration phase exposes a hierarchical symbol table, the binding phase exposes the result of the compiler's semantic analysis, and the emit phase is an API that produces IL byte codes.



Each compiler combines these components together as a single end-to-end whole.

These APIs are the same ones used by Visual Studio. For instance, the code outlining and formatting features use the syntax trees, the **Object Browser**, and navigation features use the symbol table, refactorings and **Go to Definition** use the semantic model, and **Edit and Continue** uses all of these, including the Emit API.

API layers

The .NET compiler SDK consists of several layers of APIs: compiler APIs, diagnostic APIs, scripting APIs, and workspaces APIs.

Compiler APIs

The compiler layer contains the object models that correspond to information exposed at each phase of the compiler pipeline, both syntactic and semantic. The compiler layer also contains an immutable snapshot of a single invocation of a compiler, including assembly references, compiler options, and source code files. There are two distinct APIs that represent the C# language and the Visual Basic language. The two APIs are similar in shape but tailored for high-fidelity to each individual language. This layer has no dependencies on Visual Studio components.

Diagnostic APIs

As part of its analysis, the compiler may produce a set of diagnostics covering everything from syntax, semantic, and definite assignment errors to various warnings and informational diagnostics. The Compiler API layer exposes diagnostics through an

extensible API that allows user-defined analyzers to be plugged into the compilation process. It allows user-defined diagnostics, such as those produced by tools like StyleCop, to be produced alongside compiler-defined diagnostics. Producing diagnostics in this way has the benefit of integrating naturally with tools such as MSBuild and Visual Studio, which depend on diagnostics for experiences such as halting a build based on policy and showing live squiggles in the editor and suggesting code fixes.

Scripting APIs

Hosting and scripting APIs are built on top of the compiler layer. You can use the scripting APIs to run code snippets and accumulate a runtime execution context. The C# interactive REPL (Read-Evaluate-Print Loop) uses these APIs. The REPL enables you to use C# as a scripting language, running the code interactively as you write it.

Workspaces APIs

The Workspaces layer contains the Workspace API, which is the starting point for doing code analysis and refactoring over entire solutions. It assists you in organizing all the information about the projects in a solution into a single object model, offering you direct access to the compiler layer object models without needing to parse files, configure options, or manage project-to-project dependencies.

In addition, the Workspaces layer surfaces a set of APIs used when implementing code analysis and refactoring tools that function within a host environment like the Visual Studio IDE. Examples include the Find All References, Formatting, and Code Generation APIs.

This layer has no dependencies on Visual Studio components.

Work with syntax

Article • 09/15/2021

The *syntax tree* is a fundamental immutable data structure exposed by the compiler APIs. These trees represent the lexical and syntactic structure of source code. They serve two important purposes:

- To allow tools - such as an IDE, add-ins, code analysis tools, and refactorings - to see and process the syntactic structure of source code in a user's project.
- To enable tools - such as refactorings and an IDE - to create, modify, and rearrange source code in a natural manner without having to use direct text edits. By creating and manipulating trees, tools can easily create and rearrange source code.

Syntax trees

Syntax trees are the primary structure used for compilation, code analysis, binding, refactoring, IDE features, and code generation. No part of the source code is understood without it first being identified and categorized into one of many well-known structural language elements.

> **Note**
>
> RoslynQuoter is an open-source tool that shows the syntax factory API calls used to construct a program's syntax tree. To try it out live, see http://roslynquoter.azurewebsites.net.

Syntax trees have three key attributes:

- They hold all the source information in full fidelity. Full fidelity means that the syntax tree contains every piece of information found in the source text, every grammatical construct, every lexical token, and everything else in between, including white space, comments, and preprocessor directives. For example, each literal mentioned in the source is represented exactly as it was typed. Syntax trees also capture errors in source code when the program is incomplete or malformed by representing skipped or missing tokens.
- They can produce the exact text that they were parsed from. From any syntax node, it's possible to get the text representation of the subtree rooted at that node. This ability means that syntax trees can be used as a way to construct and edit source text. By creating a tree you have, by implication, created the equivalent

text, and by making a new tree out of changes to an existing tree, you have effectively edited the text.
- They are immutable and thread-safe. After a tree is obtained, it's a snapshot of the current state of the code and never changes. This allows multiple users to interact with the same syntax tree at the same time in different threads without locking or duplication. Because the trees are immutable and no modifications can be made directly to a tree, factory methods help create and modify syntax trees by creating additional snapshots of the tree. The trees are efficient in the way they reuse underlying nodes, so a new version can be rebuilt fast and with little extra memory.

A syntax tree is literally a tree data structure, where non-terminal structural elements parent other elements. Each syntax tree is made up of nodes, tokens, and trivia.

Syntax nodes

Syntax nodes are one of the primary elements of syntax trees. These nodes represent syntactic constructs such as declarations, statements, clauses, and expressions. Each category of syntax nodes is represented by a separate class derived from Microsoft.CodeAnalysis.SyntaxNode. The set of node classes is not extensible.

All syntax nodes are non-terminal nodes in the syntax tree, which means they always have other nodes and tokens as children. As a child of another node, each node has a parent node that can be accessed through the SyntaxNode.Parent property. Because nodes and trees are immutable, the parent of a node never changes. The root of the tree has a null parent.

Each node has a SyntaxNode.ChildNodes() method, which returns a list of child nodes in sequential order based on their position in the source text. This list does not contain tokens. Each node also has methods to examine Descendants, such as DescendantNodes, DescendantTokens, or DescendantTrivia - that represent a list of all the nodes, tokens, or trivia that exist in the subtree rooted by that node.

In addition, each syntax node subclass exposes all the same children through strongly typed properties. For example, a BinaryExpressionSyntax node class has three additional properties specific to binary operators: Left, OperatorToken, and Right. The type of Left and Right is ExpressionSyntax, and the type of OperatorToken is SyntaxToken.

Some syntax nodes have optional children. For example, an IfStatementSyntax has an optional ElseClauseSyntax. If the child is not present, the property returns null.

Syntax tokens

Syntax tokens are the terminals of the language grammar, representing the smallest syntactic fragments of the code. They are never parents of other nodes or tokens. Syntax tokens consist of keywords, identifiers, literals, and punctuation.

For efficiency purposes, the SyntaxToken type is a CLR value type. Therefore, unlike syntax nodes, there is only one structure for all kinds of tokens with a mix of properties that have meaning depending on the kind of token that is being represented.

For example, an integer literal token represents a numeric value. In addition to the raw source text the token spans, the literal token has a Value property that tells you the exact decoded integer value. This property is typed as Object because it may be one of many primitive types.

The ValueText property tells you the same information as the Value property; however this property is always typed as String. An identifier in C# source text may include Unicode escape characters, yet the syntax of the escape sequence itself is not considered part of the identifier name. So although the raw text spanned by the token does include the escape sequence, the ValueText property does not. Instead, it includes the Unicode characters identified by the escape. For example, if the source text contains an identifier written as `\u03C0`, then the ValueText property for this token will return `π`.

Syntax trivia

Syntax trivia represent the parts of the source text that are largely insignificant for normal understanding of the code, such as white space, comments, and preprocessor directives. Like syntax tokens, trivia are value types. The single Microsoft.CodeAnalysis.SyntaxTrivia type is used to describe all kinds of trivia.

Because trivia are not part of the normal language syntax and can appear anywhere between any two tokens, they are not included in the syntax tree as a child of a node. Yet, because they are important when implementing a feature like refactoring and to maintain full fidelity with the source text, they do exist as part of the syntax tree.

You can access trivia by inspecting a token's SyntaxToken.LeadingTrivia or SyntaxToken.TrailingTrivia collections. When source text is parsed, sequences of trivia are associated with tokens. In general, a token owns any trivia after it on the same line up to the next token. Any trivia after that line is associated with the following token. The first token in the source file gets all the initial trivia, and the last sequence of trivia in the file is tacked onto the end-of-file token, which otherwise has zero width.

Unlike syntax nodes and tokens, syntax trivia do not have parents. Yet, because they are part of the tree and each is associated with a single token, you may access the token it is

associated with using the SyntaxTrivia.Token property.

Spans

Each node, token, or trivia knows its position within the source text and the number of characters it consists of. A text position is represented as a 32-bit integer, which is a zero-based `char` index. A TextSpan object is the beginning position and a count of characters, both represented as integers. If TextSpan has a zero length, it refers to a location between two characters.

Each node has two TextSpan properties: Span and FullSpan.

The Span property is the text span from the start of the first token in the node's subtree to the end of the last token. This span does not include any leading or trailing trivia.

The FullSpan property is the text span that includes the node's normal span, plus the span of any leading or trailing trivia.

For example:

```
C#

      if (x > 3)
      {
||        // this is bad
          |throw new Exception("Not right.");|  // better exception?||
      }
```

The statement node inside the block has a span indicated by the single vertical bars (|). It includes the characters `throw new Exception("Not right.");`. The full span is indicated by the double vertical bars (||). It includes the same characters as the span and the characters associated with the leading and trailing trivia.

Kinds

Each node, token, or trivia has a SyntaxNode.RawKind property, of type System.Int32, that identifies the exact syntax element represented. This value can be cast to a language-specific enumeration. Each language, C# or Visual Basic, has a single `SyntaxKind` enumeration (Microsoft.CodeAnalysis.CSharp.SyntaxKind and Microsoft.CodeAnalysis.VisualBasic.SyntaxKind, respectively) that lists all the possible nodes, tokens, and trivia elements in the grammar. This conversion can be done automatically by accessing the CSharpExtensions.Kind or VisualBasicExtensions.Kind extension methods.

The RawKind property allows for easy disambiguation of syntax node types that share the same node class. For tokens and trivia, this property is the only way to distinguish one type of element from another.

For example, a single BinaryExpressionSyntax class has Left, OperatorToken, and Right as children. The Kind property distinguishes whether it is an AddExpression, SubtractExpression, or MultiplyExpression kind of syntax node.

> **Tip**
>
> It's recommended to check kinds using **IsKind** (for C#) or **IsKind** (for VB) extension methods.

Errors

Even when the source text contains syntax errors, a full syntax tree that is round-trippable to the source is exposed. When the parser encounters code that does not conform to the defined syntax of the language, it uses one of two techniques to create a syntax tree:

- If the parser expects a particular kind of token but does not find it, it may insert a missing token into the syntax tree in the location that the token was expected. A missing token represents the actual token that was expected, but it has an empty span, and its SyntaxNode.IsMissing property returns `true`.
- The parser may skip tokens until it finds one where it can continue parsing. In this case, the skipped tokens are attached as a trivia node with the kind SkippedTokensTrivia.

Work with semantics

Article • 09/15/2021

Syntax trees represent the lexical and syntactic structure of source code. Although this information alone is enough to describe all the declarations and logic in the source, it is not enough information to identify what is being referenced. A name may represent:

- a type
- a field
- a method
- a local variable

Although each of these is uniquely different, determining which one an identifier actually refers to often requires a deep understanding of the language rules.

There are program elements represented in source code, and programs can also refer to previously compiled libraries, packaged in assembly files. Although no source code, and therefore no syntax nodes or trees, are available for assemblies, programs can still refer to elements inside them.

For those tasks, you need the **Semantic model**.

In addition to a syntactic model of the source code, a semantic model encapsulates the language rules, giving you an easy way to correctly match identifiers with the correct program element being referenced.

Compilation

A compilation is a representation of everything needed to compile a C# or Visual Basic program, which includes all the assembly references, compiler options, and source files.

Because all this information is in one place, the elements contained in the source code can be described in more detail. The compilation represents each declared type, member, or variable as a symbol. The compilation contains a variety of methods that help you find and relate the symbols that have either been declared in the source code or imported as metadata from an assembly.

Similar to syntax trees, compilations are immutable. After you create a compilation, it cannot be changed by you or anyone else you might be sharing it with. However, you can create a new compilation from an existing compilation, specifying a change as you do so. For example, you might create a compilation that is the same in every way as an

existing compilation, except it may include an additional source file or assembly reference.

Symbols

A symbol represents a distinct element declared by the source code or imported from an assembly as metadata. Every namespace, type, method, property, field, event, parameter, or local variable is represented by a symbol.

A variety of methods and properties on the Compilation type help you find symbols. For example, you can find a symbol for a declared type by its common metadata name. You can also access the entire symbol table as a tree of symbols rooted by the global namespace.

Symbols also contain additional information that the compiler determines from the source or metadata, such as other referenced symbols. Each kind of symbol is represented by a separate interface derived from ISymbol, each with its own methods and properties detailing the information the compiler has gathered. Many of these properties directly reference other symbols. For example, the IMethodSymbol.ReturnType property tells you the actual type symbol that the method returns.

Symbols present a common representation of namespaces, types, and members, between source code and metadata. For example, a method that was declared in source code and a method that was imported from metadata are both represented by an IMethodSymbol with the same properties.

Symbols are similar in concept to the CLR type system as represented by the System.Reflection API, yet they are richer in that they model more than just types. Namespaces, local variables, and labels are all symbols. In addition, symbols are a representation of language concepts, not CLR concepts. There is a lot of overlap, but there are many meaningful distinctions as well. For instance, an iterator method in C# or Visual Basic is a single symbol. However, when the iterator method is translated to CLR metadata, it is a type and multiple methods.

Semantic model

A semantic model represents all the semantic information for a single source file. You can use it to discover the following:

- The symbols referenced at a specific location in source.
- The resultant type of any expression.

- All diagnostics, which are errors and warnings.
- How variables flow in and out of regions of source.
- The answers to more speculative questions.

Work with a workspace

Article • 09/15/2021

The **Workspaces** layer is the starting point for doing code analysis and refactoring over entire solutions. Within this layer, the Workspace API assists you in organizing all the information about the projects in a solution into a single object model, offering you direct access to compiler layer object models like source text, syntax trees, semantic models, and compilations without needing to parse files, configure options, or manage inter-project dependencies.

Host environments, like an IDE, provide a workspace for you corresponding to the open solution. It is also possible to use this model outside of an IDE by simply loading a solution file.

Workspace

A workspace is an active representation of your solution as a collection of projects, each with a collection of documents. A workspace is typically tied to a host environment that is constantly changing as a user types or manipulates properties.

The Workspace provides access to the current model of the solution. When a change in the host environment occurs, the workspace fires corresponding events, and the Workspace.CurrentSolution property is updated. For example, when the user types in a text editor corresponding to one of the source documents, the workspace uses an event to signal that the overall model of the solution has changed and which document was modified. You can then react to those changes by analyzing the new model for correctness, highlighting areas of significance, or making a suggestion for a code change.

You can also create stand-alone workspaces that are disconnected from the host environment or used in an application that has no host environment.

Solutions, projects, and documents

Although a workspace may change every time a key is pressed, you can work with the model of the solution in isolation.

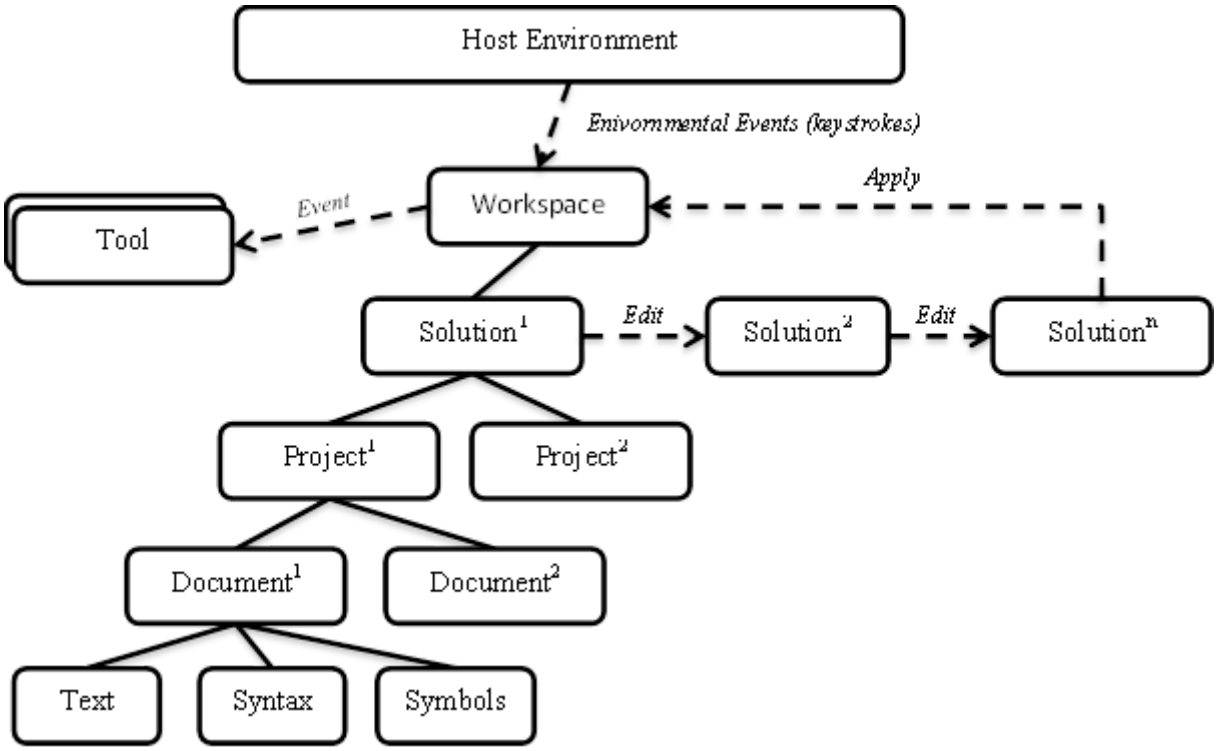
A solution is an immutable model of the projects and documents. This means that the model can be shared without locking or duplication. After you obtain a solution instance from the Workspace.CurrentSolution property, that instance will never change. However,

like with syntax trees and compilations, you can modify solutions by constructing new instances based on existing solutions and specific changes. To get the workspace to reflect your changes, you must explicitly apply the changed solution back to the workspace.

A project is a part of the overall immutable solution model. It represents all the source code documents, parse and compilation options, and both assembly and project-to-project references. From a project, you can access the corresponding compilation without needing to determine project dependencies or parse any source files.

A document is also a part of the overall immutable solution model. A document represents a single source file from which you can access the text of the file, the syntax tree, and the semantic model.

The following diagram is a representation of how the Workspace relates to the host environment, tools, and how edits are made.



Summary

Roslyn exposes a set of compiler APIs and Workspaces APIs that provides rich information about your source code and that has full fidelity with the C# and Visual Basic languages. The .NET Compiler Platform SDK dramatically lowers the barrier to entry for creating code-focused tools and applications. It creates many opportunities for innovation in areas such as meta-programming, code generation and transformation, interactive use of the C# and Visual Basic languages, and embedding of C# and Visual Basic in domain-specific languages.

Explore code with the Roslyn syntax visualizer in Visual Studio

Summarize this article for me

This article provides an overview of the Syntax Visualizer tool that ships as part of the .NET Compiler Platform ("Roslyn") SDK. The Syntax Visualizer is a tool window that helps you inspect and explore syntax trees. It's an essential tool to understand the models for code you want to analyze. It's also a debugging aid when you develop your own applications using the .NET Compiler Platform ("Roslyn") SDK. Open this tool as you create your first analyzers. The visualizer helps you understand the models used by the APIs. You can also use tools like SharpLab or LINQPad to inspect code and understand syntax trees.

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Make sure the box for **.NET Compiler Platform SDK** is checked.
6. Select **Modify**.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab

4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.
5. Select **Modify**.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

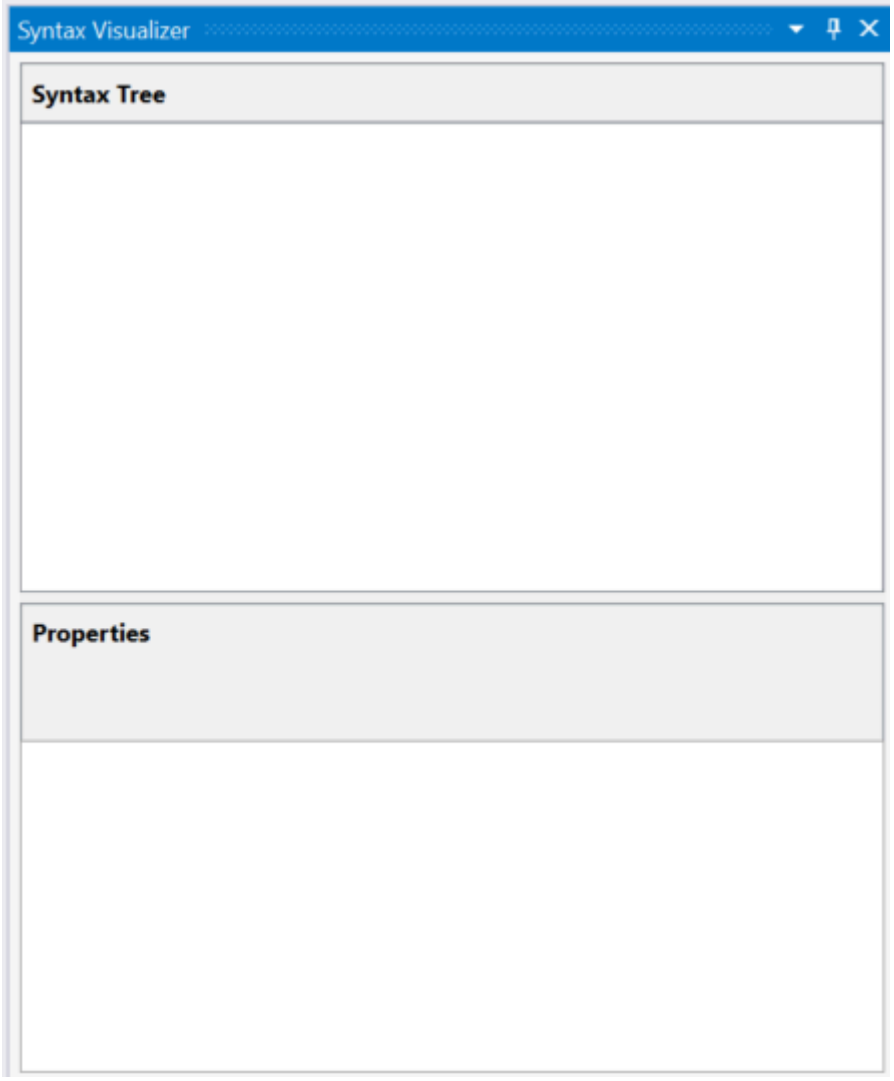
1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Familiarize yourself with the concepts used in the .NET Compiler Platform SDK by reading the overview article. It provides an introduction to syntax trees, nodes, tokens, and trivia.

Syntax Visualizer

The **Syntax Visualizer** enables inspection of the syntax tree for the C# or Visual Basic code file in the current active editor window inside the Visual Studio IDE. The visualizer can be launched by clicking on **View** > **Other Windows** > **Syntax Visualizer**. You can also use the **Quick Launch** toolbar in the upper right corner. Type "syntax", and the command to open the **Syntax Visualizer** should appear.

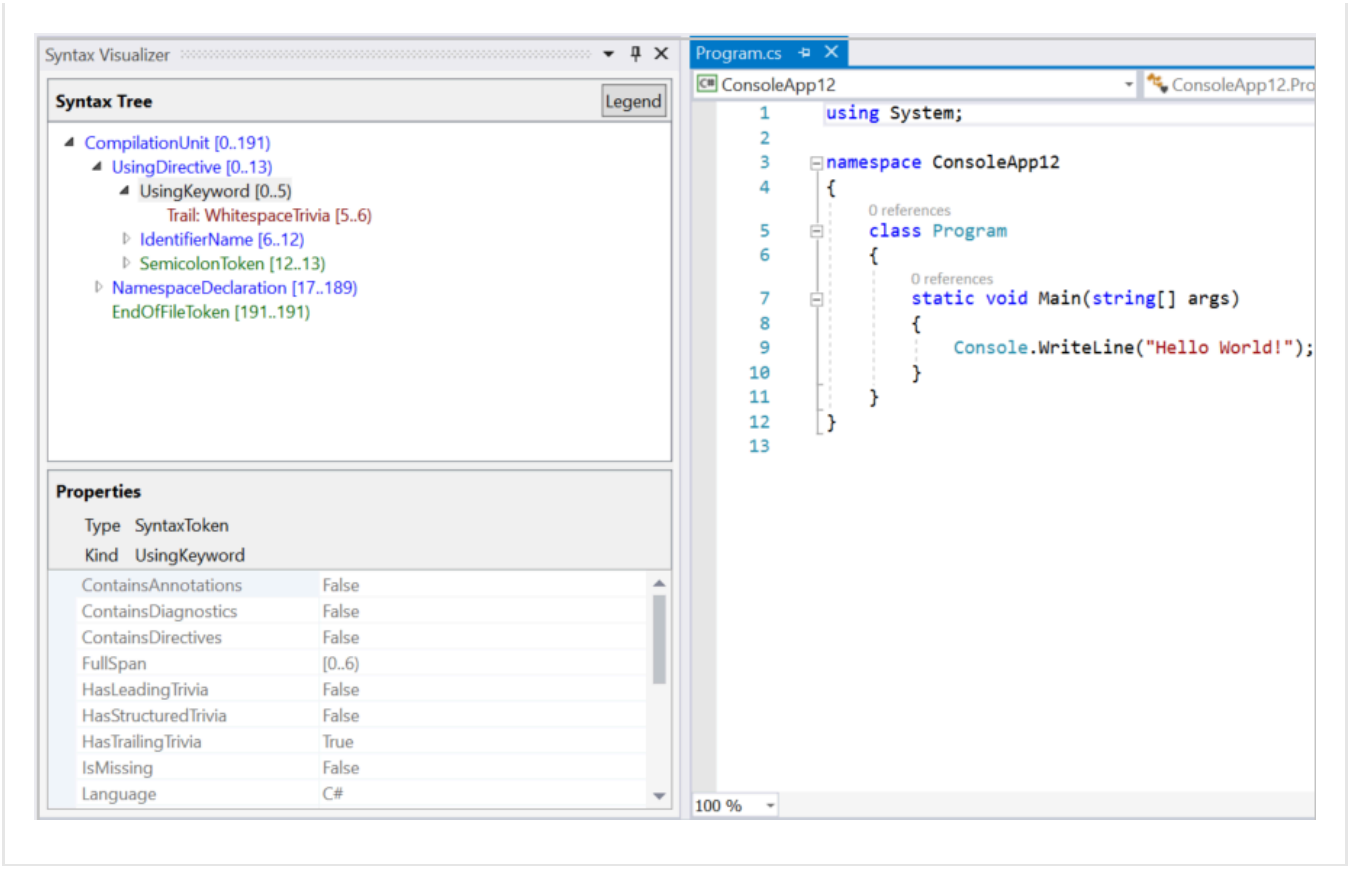
This command opens the Syntax Visualizer as a floating tool window. If you don't have a code editor window open, the display is blank, as shown in the following figure.



Dock this tool window at a convenient location inside Visual Studio, such as the left side. The Visualizer shows information about the current code file.

Create a new project using the **File** > **New Project** command. You can create either a Visual Basic or C# project. When Visual Studio opens the main code file for this project, the visualizer displays the syntax tree for it. You can open any existing C# / Visual Basic file in this Visual Studio instance, and the visualizer displays that file's syntax tree. If you have multiple code files open inside Visual Studio, the visualizer displays the syntax tree for the currently active code file, (the code file that has keyboard focus.)

C#



As shown in the preceding images, the visualizer tool window displays the syntax tree at the top and a property grid at the bottom. The property grid displays the properties of the item that is currently selected in the tree, including the .NET *Type* and the *Kind* (SyntaxKind) of the item.

Syntax trees comprise three types of items – *nodes*, *tokens*, and *trivia*. You can read more about these types in the Work with syntax article. Items of each type are represented using a different color. Click on the 'Legend' button for an overview of the colors used.

Each item in the tree also displays its own **span**. The **span** is the indices (the starting and ending position) of that node in the text file. In the preceding C# example, the selected "UsingKeyword [0..5)" token has a **Span** that is five characters wide, [0..5). The "[..)" notation means that the starting index is part of the span, but the ending index is not.

There are two ways to navigate the tree:

- Expand or click on items in the tree. The visualizer automatically selects the text corresponding to this item's span in the code editor.
- Click or select text in the code editor. In the preceding Visual Basic example, if you select the line containing "Module Module1" in the code editor, the visualizer automatically navigates to the corresponding ModuleStatement node in the tree.

The visualizer highlights the item in the tree whose span best matches the span of the text selected in the editor.

The visualizer refreshes the tree to match modifications in the active code file. Add a call to `Console.WriteLine()` inside `Main()`. As you type, the visualizer refreshes the tree.

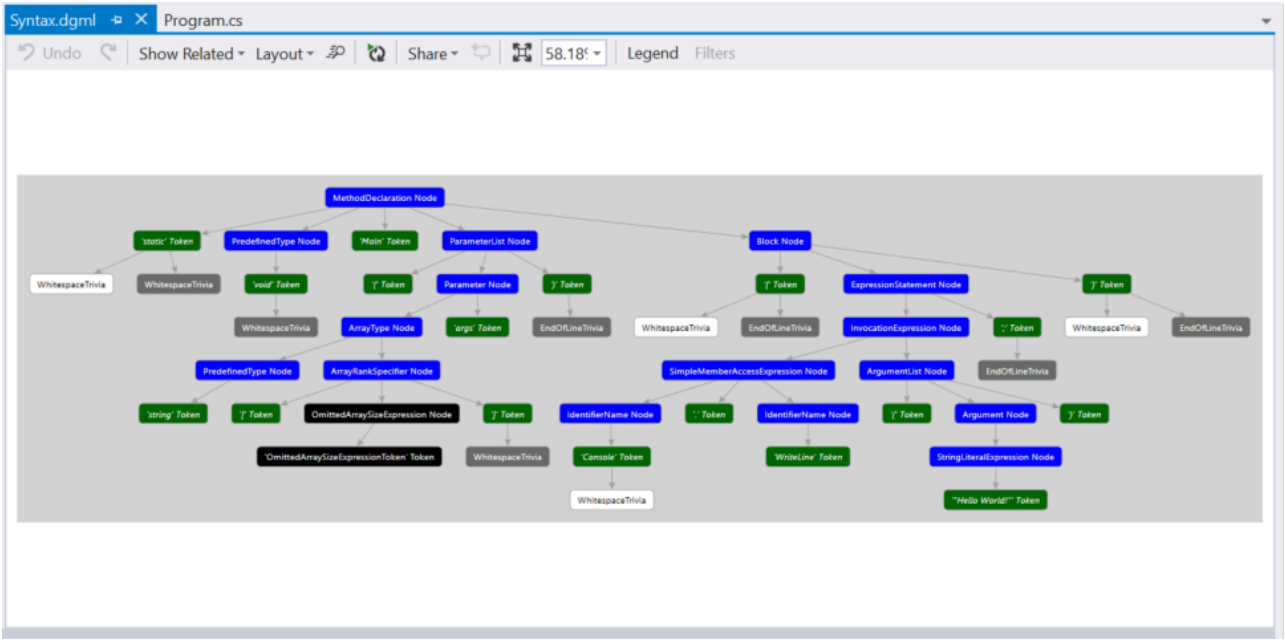
Pause typing once you have typed `Console.`. The tree has some items colored in pink. At this point, there are errors (also referred to as 'Diagnostics') in the typed code. These errors are attached to nodes, tokens, and trivia in the syntax tree. The visualizer shows you which items have errors attached to them highlighting the background in pink. You can inspect the errors on any item colored pink by hovering over the item. The visualizer only displays syntactic errors (those errors related to the syntax of the typed code); it doesn't display any semantic errors.

Syntax Graphs

Right click on any item in the tree and click on **View Directed Syntax Graph**.

C#

The visualizer displays a graphical representation of the subtree rooted at the selected item. Try these steps for the **MethodDeclaration** node corresponding to the `Main()` method in the C# example. The visualizer displays a syntax graph that looks as follows:

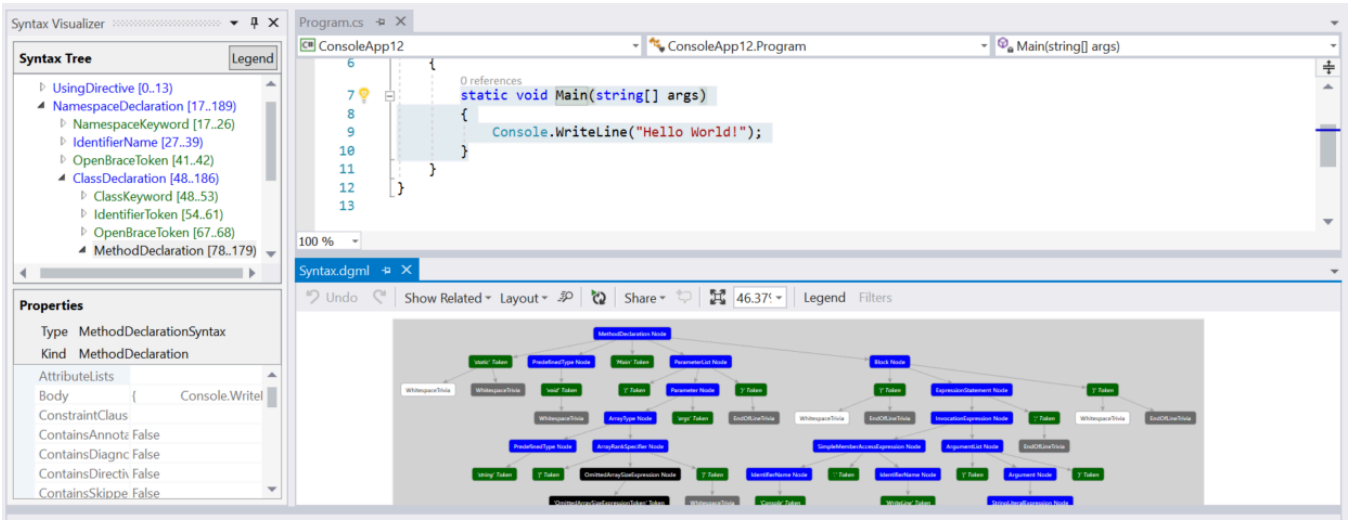


The syntax graph viewer has an option to display a legend for its coloring scheme. You can also hover over individual items in the syntax graph with the mouse to view the properties corresponding to that item.

You can view syntax graphs for different items in the tree repeatedly and the graphs will always be displayed in the same window inside Visual Studio. You can dock this window at a

convenient location inside Visual Studio so that you don't have to switch between tabs to view a new syntax graph. The bottom, below code editor windows, is often convenient.

Here is the docking layout to use with the visualizer tool window and the syntax graph window:

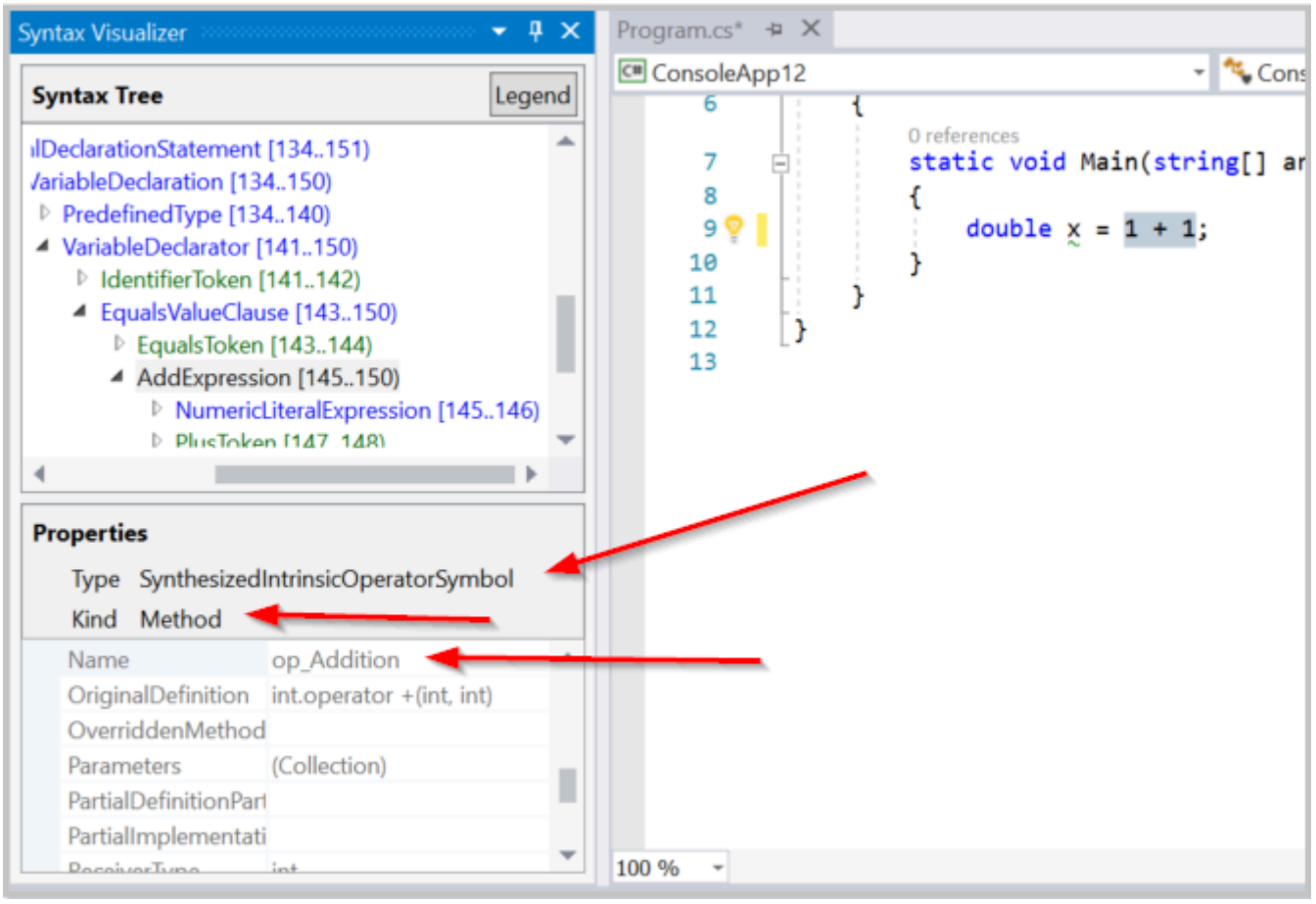


Another option is to put the syntax graph window on a second monitor, in a dual monitor setup.

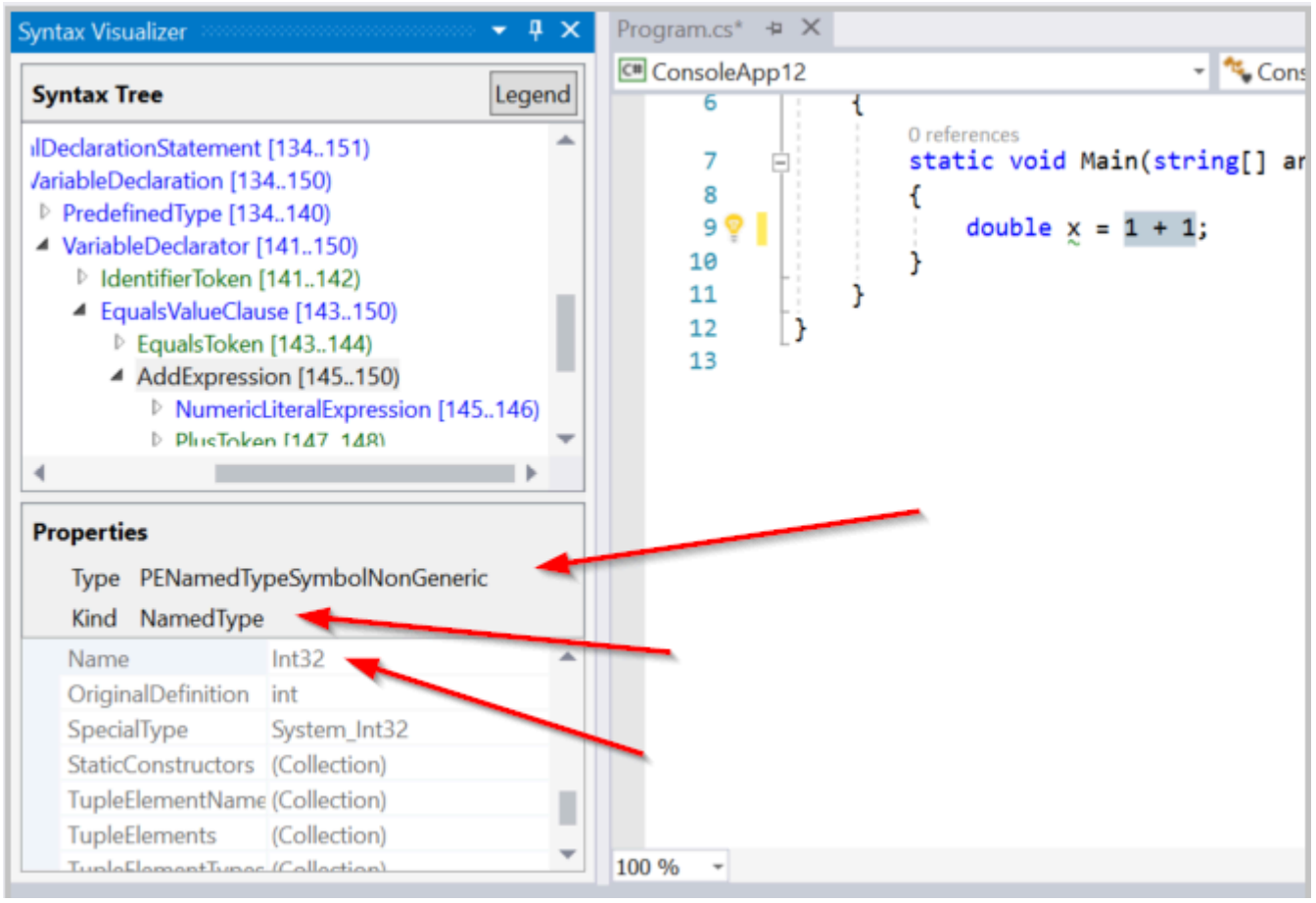
Inspecting semantics

The Syntax Visualizer enables rudimentary inspection of symbols and semantic information. Type `double x = 1 + 1;` inside Main() in the C# example. Then, select the expression `1 + 1` in the code editor window. The visualizer highlights the **AddExpression** node in the visualizer. Right click on this **AddExpression** and click on **View Symbol (if any)**. Notice that most of the menu items have the "if any" qualifier. The Syntax Visualizer inspects properties of a Node, including properties that may not be present for all nodes.

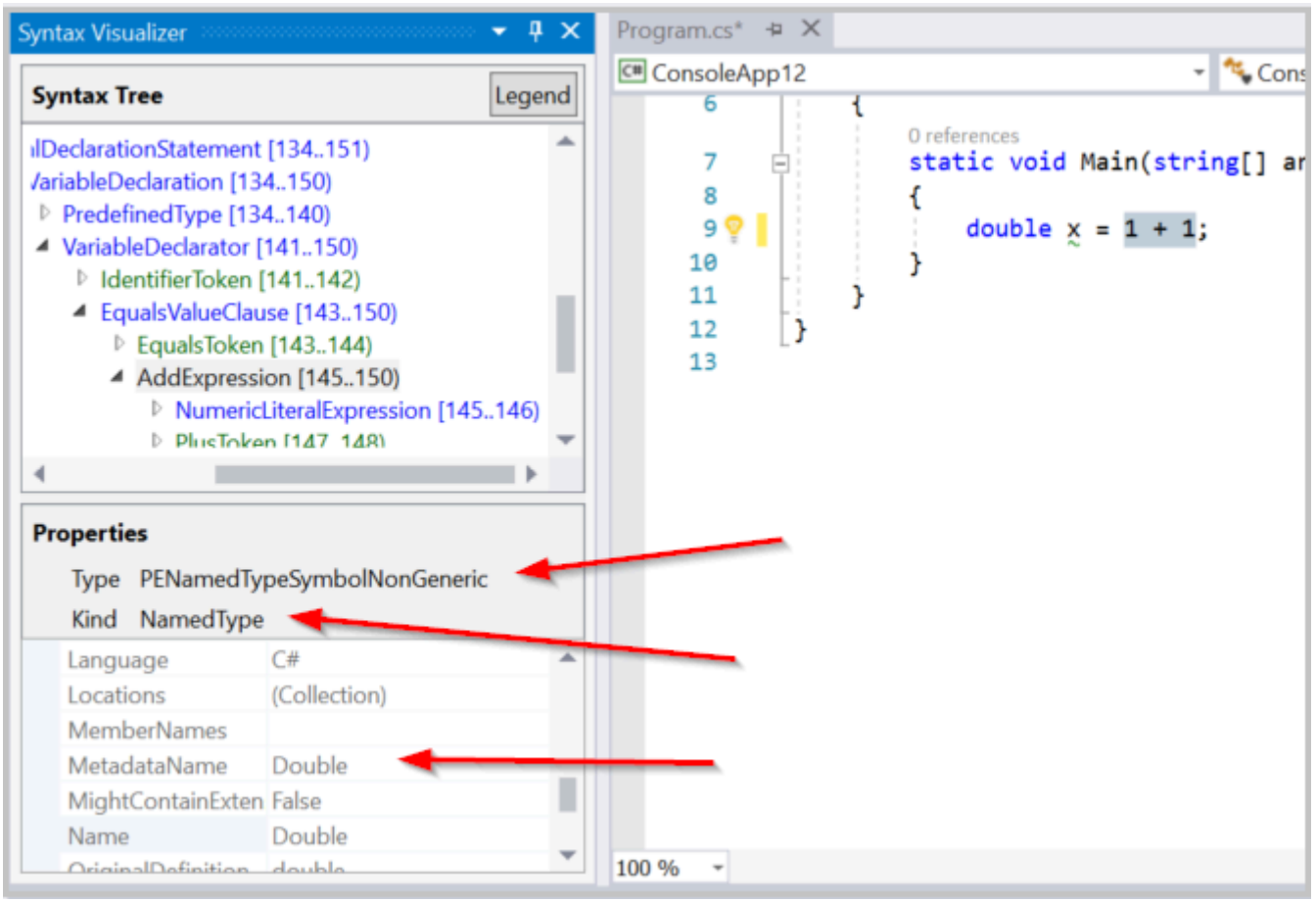
The property grid in the visualizer updates as shown in the following figure: The symbol for the expression is a **SynthesizedIntrinsicOperatorSymbol** with **Kind = Method**.



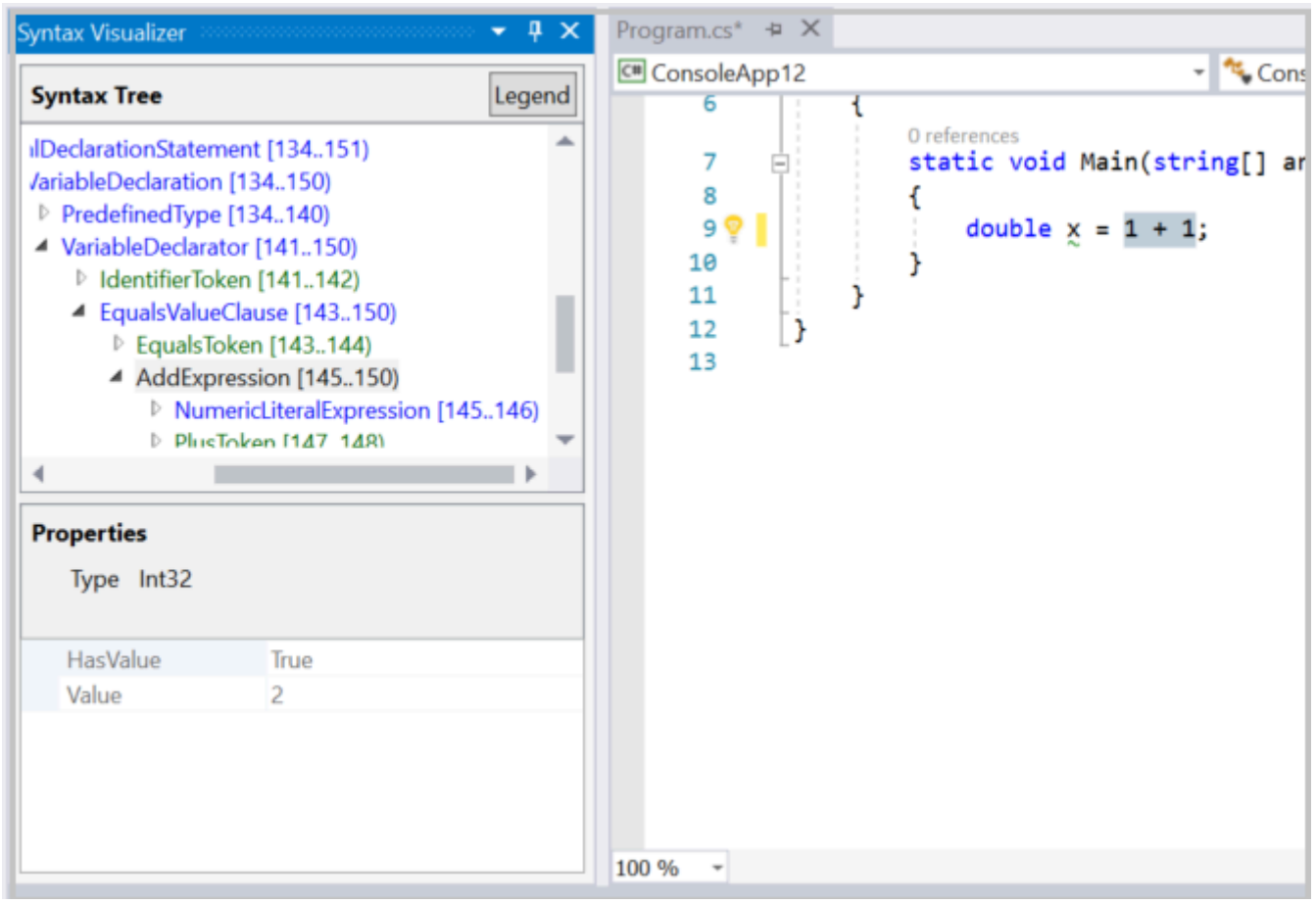
Right-click the same **AddExpression** node and select **View TypeSymbol (if any)**. The property grid in the visualizer updates as shown in the following figure, indicating that the type of the selected expression is `Int32`.



Right-click the same **AddExpression** node and select **View Converted TypeSymbol (if any)**. The property grid updates indicating that although the type of the expression is `Int32`, the converted type of the expression is `Double` as shown in the following figure. This node includes converted type symbol information because the `Int32` expression occurs in a context where it must be converted to a `Double`. This conversion satisfies the `Double` type specified for the variable `x` on the left-hand side of the assignment operator.



Finally, right-click the same **AddExpression** node and select **View Constant Value (if any)**. The property grid shows that the value of the expression is a compile time constant with value `2`.



The preceding example can also be replicated in Visual Basic. Type `Dim x As Double = 1 + 1` in a Visual Basic file. Select the expression `1 + 1` in the code editor window. The visualizer highlights the corresponding **AddExpression** node in the visualizer. Repeat the preceding steps for this **AddExpression** and you should see identical results.

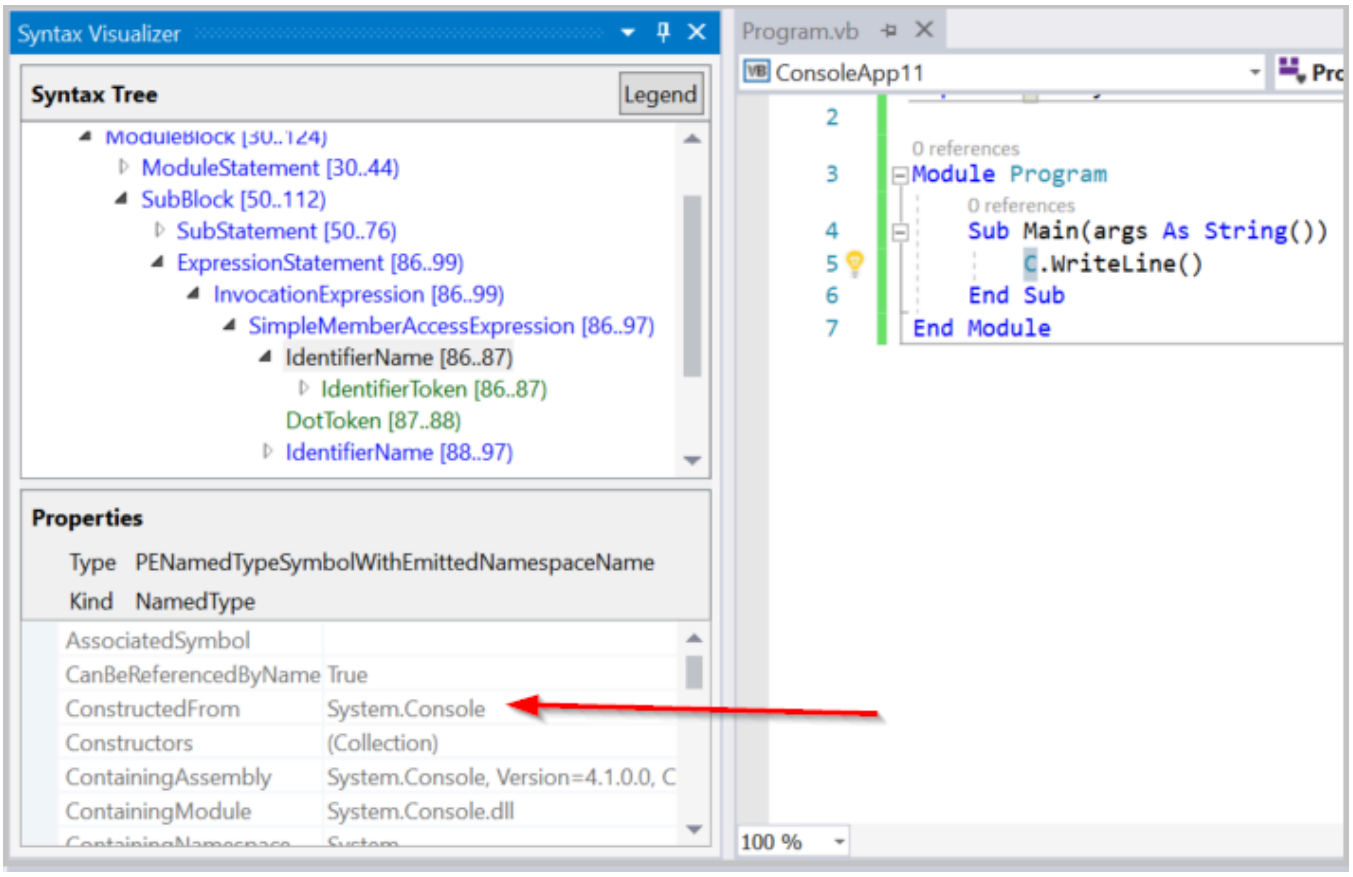
Examine more code in Visual Basic. Update your main Visual Basic file with the following code:

VB

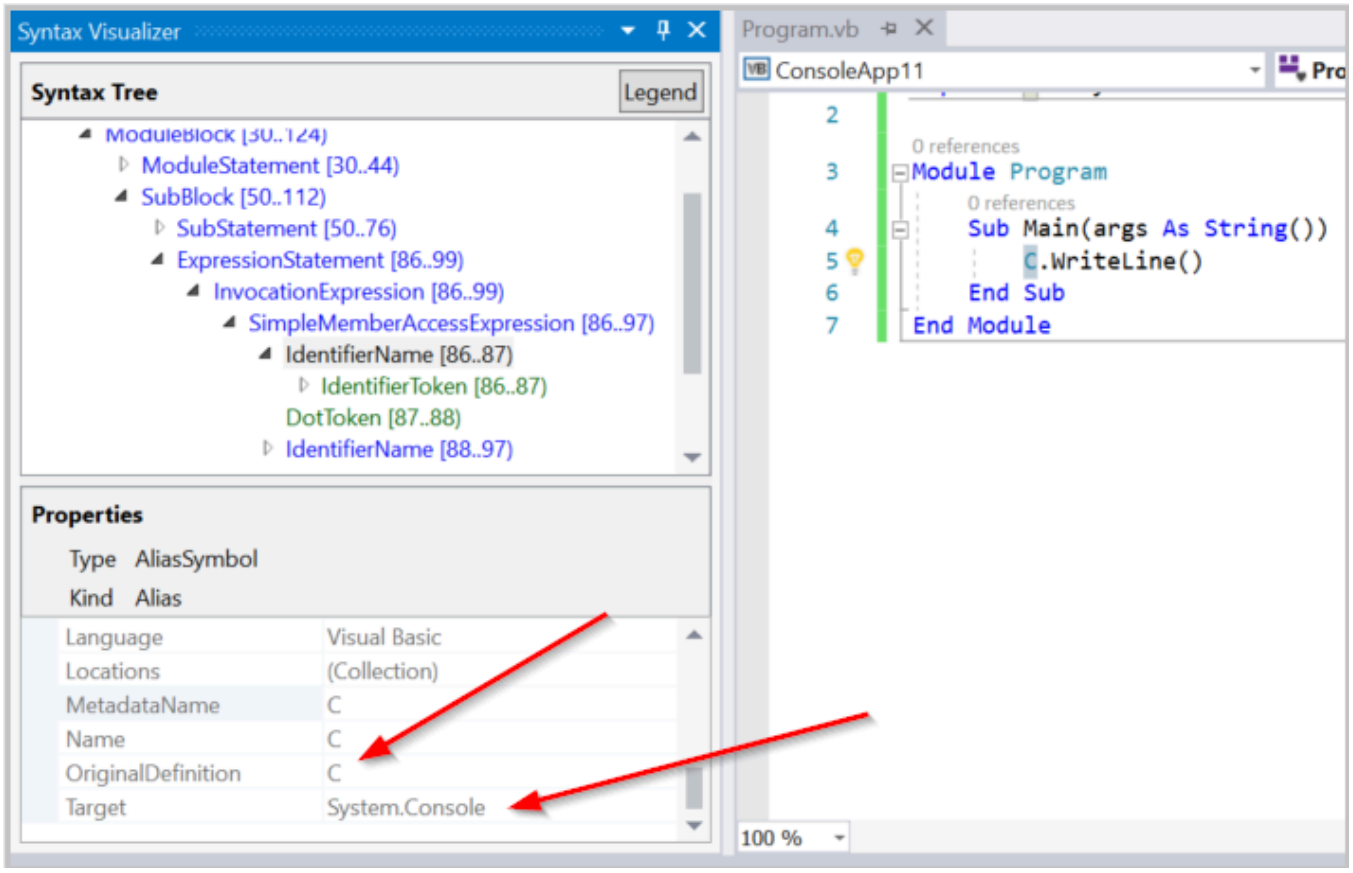
```
Imports C = System.Console

Module Program
    Sub Main(args As String())
        C.WriteLine()
    End Sub
End Module
```

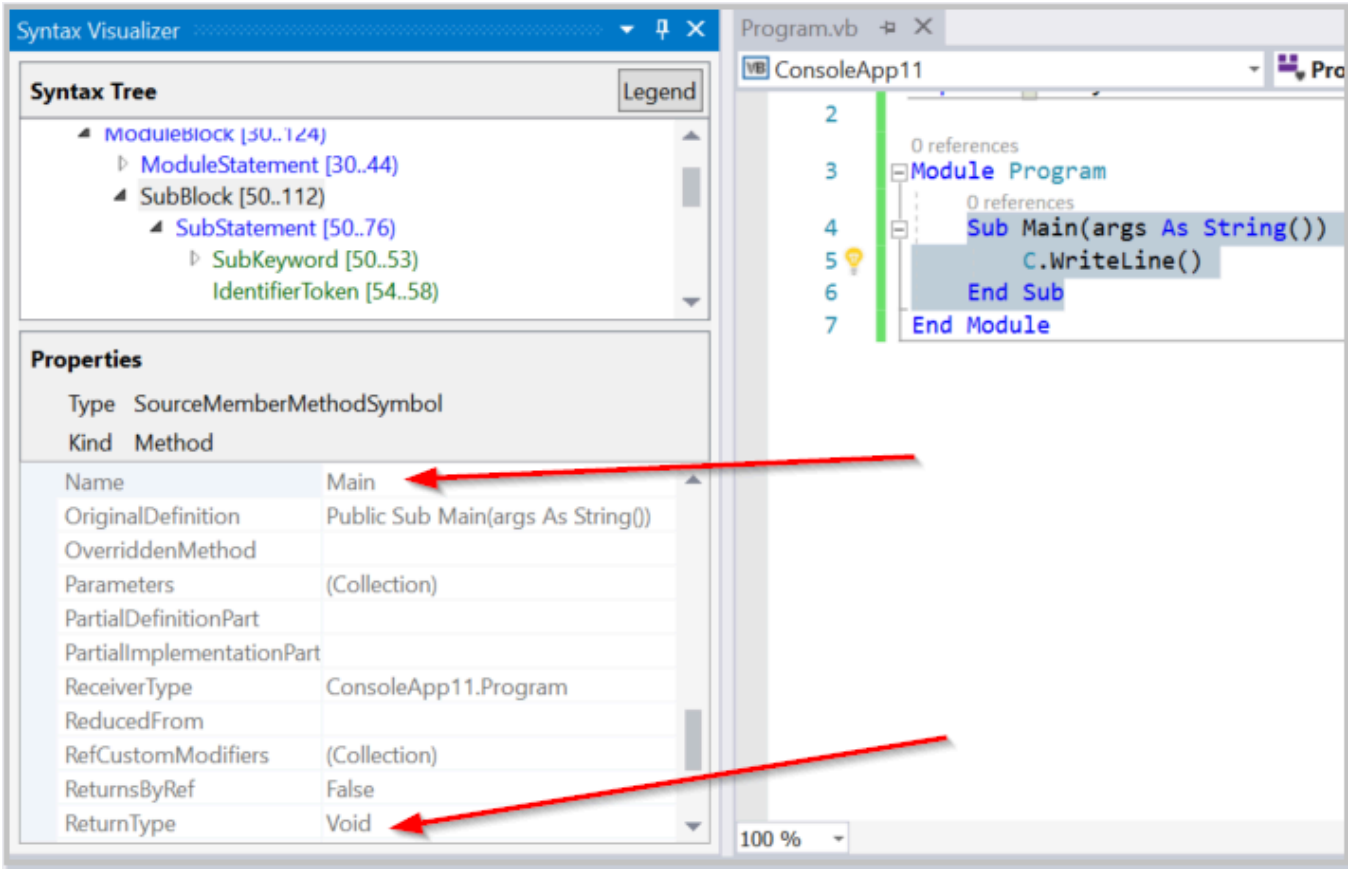
This code introduces an alias named `C` that maps to the type `System.Console` at the top of the file and uses this alias inside `Main()`. Select the use of this alias, the `C` in `C.WriteLine()`, inside the `Main()` method. The visualizer selects the corresponding **IdentifierName** node in the visualizer. Right-click this node and click on **View Symbol (if any)**. The property grid indicates that this identifier is bound to the type `System.Console` as shown in the following figure:



Right-click the same **IdentifierName** node and select **View AliasSymbol (if any)**. The property grid indicates the identifier is an alias with name `C` that is bound to the `System.Console` target. In other words, the property grid provides information regarding the **AliasSymbol** corresponding to the identifier `C`.



Inspect the symbol corresponding to any declared type, method, property. Select the corresponding node in the visualizer and right-click to access **View Symbol (if any)**. Select the method `Sub Main()`, including the body of the method. Right-click the corresponding **SubBlock** node in the visualizer and select **View Symbol (if any)**. The property grid shows the **MethodSymbol** for this **SubBlock** has name `Main` with return type `Void`.



The above Visual Basic examples can be easily replicated in C#. Type `using C = System.Console;` in place of `Imports C = System.Console` for the alias. The preceding steps in C# yield identical results in the visualizer window.

Semantic inspection operations are only available on nodes. They are not available on tokens or trivia. Not all nodes have interesting semantic information to inspect. When a node doesn't have interesting semantic information, right-clicking and selecting **View * Symbol (if any)** shows a blank property grid.

You can read more about APIs for performing semantic analysis in the Work with semantics overview document.

Closing the syntax visualizer

You can close the visualizer window when you are not using it to examine source code. The syntax visualizer updates its display as you navigate through code, editing and changing the source. It can get distracting when you are not using it.

Last updated on 07/09/2025

Choose diagnostic IDs

Article • 11/03/2023

A diagnostic ID is the string associated with a given diagnostic, such as a compiler error or a diagnostic that is produced by an analyzer.

The IDs are surfaced from various APIs, such as:

- DiagnosticDescriptor.Id
- ObsoleteAttribute.DiagnosticId
- ExperimentalAttribute.DiagnosticId

Diagnostic IDs are also used as identifiers in source, for example, from #pragma warning disable or .editorconfig files.

Considerations

- Diagnostic IDs should be unique
- Diagnostic IDs must be legal identifiers in C#
- Diagnostic IDs should be less than 15 characters long
- Diagnostic IDs should be of the form `<PREFIX><number>`
  - The prefix is specific to your project
  - The number represents the specific diagnostic

> **Note**
>
> It's a source breaking change to change diagnostic IDs, as existing suppressions would be ignored if the ID changed.

Don't limit your prefix to two-characters (such as `CSXXX` and `CAXXXX`). Instead, use a longer prefix to avoid conflicts. For example, the `System.*` diagnostics use `SYSLIB` as their prefix.

Get started with syntax analysis

Article • 09/15/2021

In this tutorial, you'll explore the **Syntax API**. The Syntax API provides access to the data structures that describe a C# or Visual Basic program. These data structures have enough detail that they can fully represent any program of any size. These structures can describe complete programs that compile and run correctly. They can also describe incomplete programs, as you write them, in the editor.

To enable this rich expression, the data structures and APIs that make up the Syntax API are necessarily complex. Let's start with what the data structure looks like for the typical "Hello World" program:

C#

```
using System;
using System.Collections.Generic;
using System.Linq;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine("Hello World!");
        }
    }
}
```

Look at the text of the previous program. You recognize familiar elements. The entire text represents a single source file, or a **compilation unit**. The first three lines of that source file are **using directives**. The remaining source is contained in a **namespace declaration**. The namespace declaration contains a child **class declaration**. The class declaration contains one **method declaration**.

The Syntax API creates a tree structure with the root representing the compilation unit. Nodes in the tree represent the `using` directives, namespace declaration and all the other elements of the program. The tree structure continues down to the lowest levels: the string "Hello World!" is a **string literal token** that is a descendent of an **argument**. The Syntax API provides access to the structure of the program. You can query for specific code practices, walk the entire tree to understand the code, and create new trees by modifying the existing tree.

That brief description provides an overview of the kind of information accessible using the Syntax API. The Syntax API is nothing more than a formal API that describes the familiar code constructs you know from C#. The full capabilities include information about how the code is formatted including line breaks, white space, and indenting. Using this information, you can fully represent the code as written and read by human programmers or the compiler. Using this structure enables you to interact with the source code on a deeply meaningful level. It's no longer text strings, but data that represents the structure of a C# program.

To get started, you'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Understanding syntax trees

You use the Syntax API for any analysis of the structure of C# code. The **Syntax API** exposes the parsers, the syntax trees, and utilities for analyzing and constructing syntax trees. It's how you search code for specific syntax elements or read the code for a program.

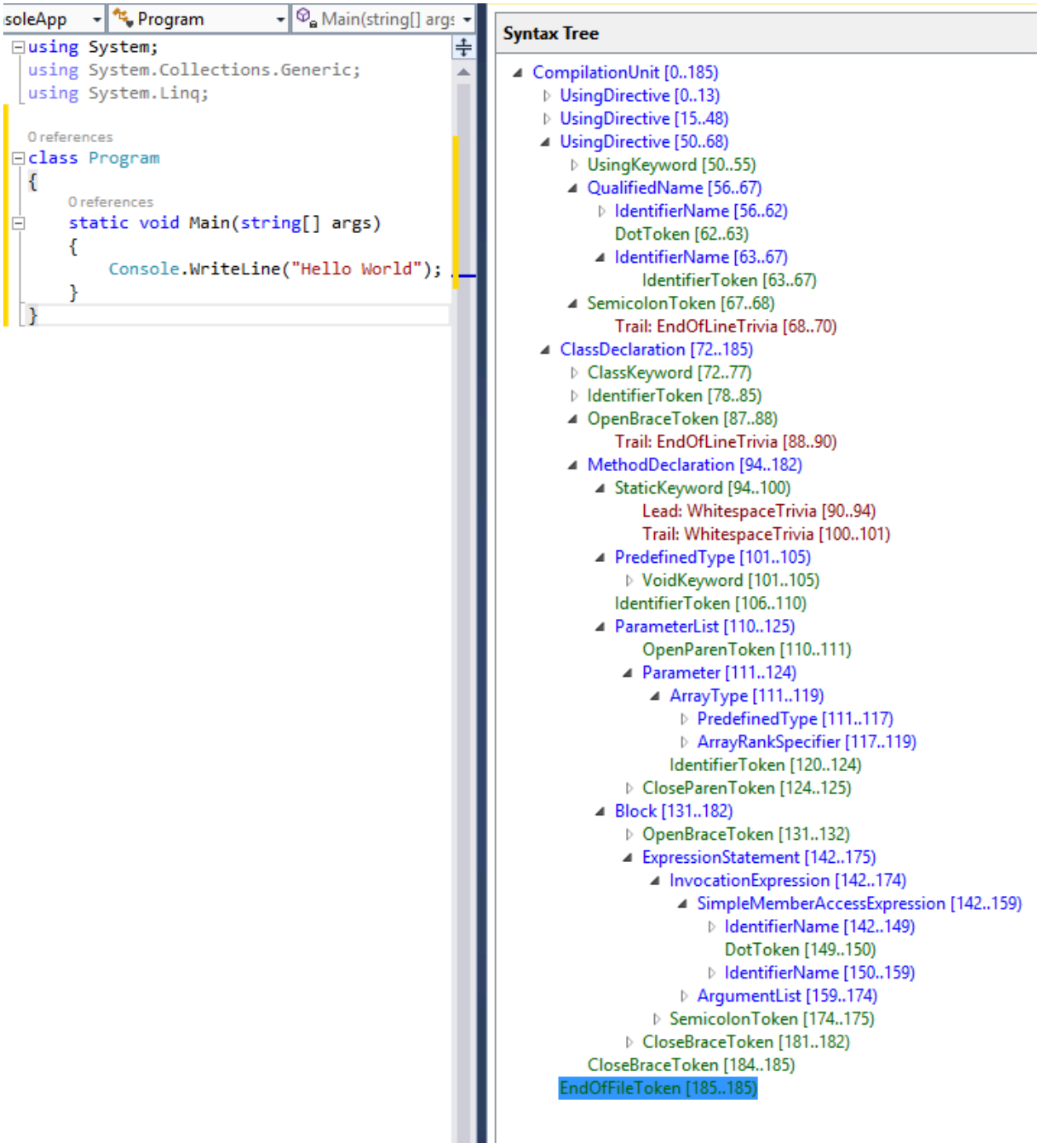
A syntax tree is a data structure used by the C# and Visual Basic compilers to understand C# and Visual Basic programs. Syntax trees are produced by the same parser that runs when a project is built or a developer hits F5. The syntax trees have full-fidelity with the language; every bit of information in a code file is represented in the tree. Writing a syntax tree to text reproduces the exact original text that was parsed. The syntax trees are also **immutable**; once created a syntax tree can never be changed. Consumers of the trees can analyze the trees on multiple threads, without locks or other concurrency measures, knowing the data never changes. You can use APIs to create new trees that are the result of modifying an existing tree.

The four primary building blocks of syntax trees are:

- The Microsoft.CodeAnalysis.SyntaxTree class, an instance of which represents an entire parse tree. SyntaxTree is an abstract class that has language-specific derivatives. You use the parse methods of the Microsoft.CodeAnalysis.CSharp.CSharpSyntaxTree (or Microsoft.CodeAnalysis.VisualBasic.VisualBasicSyntaxTree) class to parse text in C# (or Visual Basic).
- The Microsoft.CodeAnalysis.SyntaxNode class, instances of which represent syntactic constructs such as declarations, statements, clauses, and expressions.
- The Microsoft.CodeAnalysis.SyntaxToken structure, which represents an individual keyword, identifier, operator, or punctuation.
- And lastly the Microsoft.CodeAnalysis.SyntaxTrivia structure, which represents syntactically insignificant bits of information such as the white space between tokens, preprocessing directives, and comments.

Trivia, tokens, and nodes are composed hierarchically to form a tree that completely represents everything in a fragment of Visual Basic or C# code. You can see this structure using the **Syntax Visualizer** window. In Visual Studio, choose **View** > **Other Windows** > **Syntax Visualizer**. For example, the preceding C# source file examined using the **Syntax Visualizer** looks like the following figure:

SyntaxNode: Blue | **SyntaxToken**: Green | **SyntaxTrivia**: Red



By navigating this tree structure, you can find any statement, expression, token, or bit of white space in a code file.

While you can find anything in a code file using the Syntax APIs, most scenarios involve examining small snippets of code, or searching for particular statements or fragments. The two examples that follow show typical uses to browse the structure of code, or search for single statements.

Traversing trees

You can examine the nodes in a syntax tree in two ways. You can traverse the tree to examine each node, or you can query for specific elements or nodes.

Manual traversal

You can see the finished code for this sample in our GitHub repository.

> **Note**
>
> The Syntax Tree types use inheritance to describe the different syntax elements that are valid at different locations in the program. Using these APIs often means casting properties or collection members to specific derived types. In the following examples, the assignment and the casts are separate statements, using explicitly typed variables. You can read the code to see the return types of the API and the runtime type of the objects returned. In practice, it's more common to use implicitly typed variables and rely on API names to describe the type of objects being examined.

Create a new C# **Stand-Alone Code Analysis Tool** project:

- In Visual Studio, choose **File** > **New** > **Project** to display the New Project dialog.
- Under **Visual C#** > **Extensibility**, choose **Stand-Alone Code Analysis Tool**.
- Name your project "**SyntaxTreeManualTraversal**" and click OK.

You're going to analyze the basic "Hello World!" program shown earlier. Add the text for the Hello World program as a constant in your `Program` class:

C#

```
        const string programText =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

Next, add the following code to build the **syntax tree** for the code text in the `programText` constant. Add the following line to your `Main` method:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Those two lines create the tree and retrieve the root node of that tree. You can now examine the nodes in the tree. Add these lines to your `Main` method to display some of the properties of the root node in the tree:

C#

```
WriteLine($"The tree is a {root.Kind()} node.");
WriteLine($"The tree has {root.Members.Count} elements in it.");
WriteLine($"The tree has {root.Usings.Count} using directives. They are:");
foreach (UsingDirectiveSyntax element in root.Usings)
    WriteLine($"\t{element.Name}");
```

Run the application to see what your code has discovered about the root node in this tree.

Typically, you'd traverse the tree to learn about the code. In this example, you're analyzing code you know to explore the APIs. Add the following code to examine the first member of the `root` node:

C#

```
MemberDeclarationSyntax firstMember = root.Members[0];
WriteLine($"The first member is a {firstMember.Kind()}.");
var helloWorldDeclaration = (NamespaceDeclarationSyntax)firstMember;
```

That member is a Microsoft.CodeAnalysis.CSharp.Syntax.NamespaceDeclarationSyntax. It represents everything in the scope of the `namespace HelloWorld` declaration. Add the following code to examine what nodes are declared inside the `HelloWorld` namespace:

C#

```
WriteLine($"There are {helloWorldDeclaration.Members.Count} members declared
in this namespace.");
WriteLine($"The first member is a
{helloWorldDeclaration.Members[0].Kind()}.");
```

Run the program to see what you've learned.

Now that you know the declaration is a Microsoft.CodeAnalysis.CSharp.Syntax.ClassDeclarationSyntax, declare a new variable of

that type to examine the class declaration. This class only contains one member: the `Main` method. Add the following code to find the `Main` method, and cast it to a Microsoft.CodeAnalysis.CSharp.Syntax.MethodDeclarationSyntax.

C#

```
var programDeclaration =
(ClassDeclarationSyntax)helloWorldDeclaration.Members[0];
WriteLine($"There are {programDeclaration.Members.Count} members declared in
the {programDeclaration.Identifier} class.");
WriteLine($"The first member is a {programDeclaration.Members[0].Kind()}.");
var mainDeclaration =
(MethodDeclarationSyntax)programDeclaration.Members[0];
```

The method declaration node contains all the syntactic information about the method. Let's display the return type of the `Main` method, the number and types of the arguments, and the body text of the method. Add the following code:

C#

```
WriteLine($"The return type of the {mainDeclaration.Identifier} method is
{mainDeclaration.ReturnType}.");
WriteLine($"The method has {mainDeclaration.ParameterList.Parameters.Count}
parameters.");
foreach (ParameterSyntax item in mainDeclaration.ParameterList.Parameters)
    WriteLine($"The type of the {item.Identifier} parameter is
{item.Type}.");
WriteLine($"The body text of the {mainDeclaration.Identifier} method
follows:");
WriteLine(mainDeclaration.Body?.ToFullString());

var argsParameter = mainDeclaration.ParameterList.Parameters[0];
```

Run the program to see all the information you've discovered about this program:

text

```
The tree is a CompilationUnit node.
The tree has 1 elements in it.
The tree has 4 using directives. They are:
        System
        System.Collections
        System.Linq
        System.Text
The first member is a NamespaceDeclaration.
There are 1 members declared in this namespace.
The first member is a ClassDeclaration.
There are 1 members declared in the Program class.
The first member is a MethodDeclaration.
```

```
The return type of the Main method is void.
The method has 1 parameters.
The type of the args parameter is string[].
The body text of the Main method follows:
        {
            Console.WriteLine("Hello, World!");
        }
```

Query methods

In addition to traversing trees, you can also explore the syntax tree using the query methods defined on Microsoft.CodeAnalysis.SyntaxNode. These methods should be immediately familiar to anyone familiar with XPath. You can use these methods with LINQ to quickly find things in a tree. The SyntaxNode has query methods such as DescendantNodes, AncestorsAndSelf and ChildNodes.

You can use these query methods to find the argument to the `Main` method as an alternative to navigating the tree. Add the following code to the bottom of your `Main` method:

C#

```
var firstParameters = from methodDeclaration in root.DescendantNodes()
                                          .OfType<MethodDeclarationSyntax>()
                      where methodDeclaration.Identifier.ValueText == "Main"
                      select
methodDeclaration.ParameterList.Parameters.First();

var argsParameter2 = firstParameters.Single();

WriteLine(argsParameter == argsParameter2);
```

The first statement uses a LINQ expression and the DescendantNodes method to locate the same parameter as in the previous example.

Run the program, and you can see that the LINQ expression found the same parameter as manually navigating the tree.

The sample uses `WriteLine` statements to display information about the syntax trees as they are traversed. You can also learn much more by running the finished program under the debugger. You can examine more of the properties and methods that are part of the syntax tree created for the hello world program.

Syntax walkers

Often you want to find all nodes of a specific type in a syntax tree, for example, every property declaration in a file. By extending the Microsoft.CodeAnalysis.CSharp.CSharpSyntaxWalker class and overriding the VisitPropertyDeclaration(PropertyDeclarationSyntax) method, you process every property declaration in a syntax tree without knowing its structure beforehand. CSharpSyntaxWalker is a specific kind of CSharpSyntaxVisitor that recursively visits a node and each of its children.

This example implements a CSharpSyntaxWalker that examines a syntax tree. It collects `using` directives it finds that aren't importing a `System` namespace.

Create a new C# **Stand-Alone Code Analysis Tool** project; name it "**SyntaxWalker**."

You can see the finished code for this sample in our GitHub repository. The sample on GitHub contains both projects described in this tutorial.

As in the previous sample, you can define a string constant to hold the text of the program you're going to analyze:

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;

namespace TopLevel
{
    using Microsoft;
    using System.ComponentModel;

    namespace Child1
    {
        using Microsoft.Win32;
        using System.Runtime.InteropServices;

        class Foo { }
    }

    namespace Child2
    {
        using System.CodeDom;
        using Microsoft.CSharp;

        class Bar { }
```

```
    }
}";
```

This source text contains `using` directives scattered across four different locations: the file-level, in the top-level namespace, and in the two nested namespaces. This example highlights a core scenario for using the CSharpSyntaxWalker class to query code. It would be cumbersome to visit every node in the root syntax tree to find using declarations. Instead, you create a derived class and override the method that gets called only when the current node in the tree is a `using` directive. Your visitor does not do any work on any other node types. This single method examines each of the `using` directives and builds a collection of the namespaces that aren't in the `System` namespace. You build a CSharpSyntaxWalker that examines all the `using` directives, but only the `using` directives.

Now that you've defined the program text, you need to create a `SyntaxTree` and get the root of that tree:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);
CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Next, create a new class. In Visual Studio, choose **Project** > **Add New Item**. In the **Add New Item** dialog type *UsingCollector.cs* as the filename.

You implement the `using` visitor functionality in the `UsingCollector` class. Start by making the `UsingCollector` class derive from CSharpSyntaxWalker.

C#

```
class UsingCollector : CSharpSyntaxWalker
```

You need storage to hold the namespace nodes that you're collecting. Declare a public read-only property in the `UsingCollector` class; you use this variable to store the UsingDirectiveSyntax nodes you find:

C#

```
public ICollection<UsingDirectiveSyntax> Usings { get; } = new
List<UsingDirectiveSyntax>();
```

The base class, CSharpSyntaxWalker implements the logic to visit each node in the syntax tree. The derived class overrides the methods called for the specific nodes you're interested in. In this case, you're interested in any `using` directive. That means you must override the VisitUsingDirective(UsingDirectiveSyntax) method. The one argument to this method is a Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax object. That's an important advantage to using the visitors: they call the overridden methods with arguments already cast to the specific node type. The Microsoft.CodeAnalysis.CSharp.Syntax.UsingDirectiveSyntax class has a Name property that stores the name of the namespace being imported. It is a Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax. Add the following code in the VisitUsingDirective(UsingDirectiveSyntax) override:

C#

```
public override void VisitUsingDirective(UsingDirectiveSyntax node)
{
    WriteLine($"\tVisitUsingDirective called with {node.Name}.");
    if (node.Name.ToString() != "System" &&
        !node.Name.ToString().StartsWith("System."))
    {
        WriteLine($"\t\tSuccess. Adding {node.Name}.");
        this.Usings.Add(node);
    }
}
```

As with the earlier example, you've added a variety of `WriteLine` statements to aid in understanding of this method. You can see when it's called, and what arguments are passed to it each time.

Finally, you need to add two lines of code to create the `UsingCollector` and have it visit the root node, collecting all the `using` directives. Then, add a `foreach` loop to display all the `using` directives your collector found:

C#

```
var collector = new UsingCollector();
collector.Visit(root);
foreach (var directive in collector.Usings)
{
    WriteLine(directive.Name);
}
```

Compile and run the program. You should see the following output:

Console

```
        VisitUsingDirective called with System.
        VisitUsingDirective called with System.Collections.Generic.
        VisitUsingDirective called with System.Linq.
        VisitUsingDirective called with System.Text.
        VisitUsingDirective called with Microsoft.CodeAnalysis.
                Success. Adding Microsoft.CodeAnalysis.
        VisitUsingDirective called with Microsoft.CodeAnalysis.CSharp.
                Success. Adding Microsoft.CodeAnalysis.CSharp.
        VisitUsingDirective called with Microsoft.
                Success. Adding Microsoft.
        VisitUsingDirective called with System.ComponentModel.
        VisitUsingDirective called with Microsoft.Win32.
                Success. Adding Microsoft.Win32.
        VisitUsingDirective called with System.Runtime.InteropServices.
        VisitUsingDirective called with System.CodeDom.
        VisitUsingDirective called with Microsoft.CSharp.
                Success. Adding Microsoft.CSharp.
Microsoft.CodeAnalysis
Microsoft.CodeAnalysis.CSharp
Microsoft
Microsoft.Win32
Microsoft.CSharp
Press any key to continue . . .
```

Congratulations! You've used the **Syntax API** to locate specific kinds of directives and declarations in C# source code.

Get started with semantic analysis

Article • 09/15/2021

This tutorial assumes you're familiar with the Syntax API. The get started with syntax analysis article provides sufficient introduction.

In this tutorial, you explore the **Symbol** and **Binding APIs**. These APIs provide information about the *semantic meaning* of a program. They enable you to ask and answer questions about the types represented by any symbol in your program.

You'll need to install the **.NET Compiler Platform SDK**:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab

4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Understanding Compilations and Symbols

As you work more with the .NET Compiler SDK, you become familiar with the distinctions between Syntax API and the Semantic API. The **Syntax API** allows you to look at the *structure* of a program. However, often you want richer information about the semantics or *meaning* of a program. While a loose code file or snippet of Visual Basic or C# code can be syntactically analyzed in isolation, it's not meaningful to ask questions such as "what's the type of this variable" in a vacuum. The meaning of a type name may be dependent on assembly references, namespace imports, or other code files. Those questions are answered using the **Semantic API**, specifically the Microsoft.CodeAnalysis.Compilation class.

An instance of Compilation is analogous to a single project as seen by the compiler and represents everything needed to compile a Visual Basic or C# program. The **compilation** includes the set of source files to be compiled, assembly references, and compiler options. You can reason about the meaning of the code using all the other information in this context. A Compilation allows you to find **Symbols** - entities such as types, namespaces, members, and variables which names and other expressions refer to. The process of associating names and expressions with **Symbols** is called **Binding**.

Like Microsoft.CodeAnalysis.SyntaxTree, Compilation is an abstract class with language-specific derivatives. When creating an instance of Compilation, you must invoke a factory method on the Microsoft.CodeAnalysis.CSharp.CSharpCompilation (or Microsoft.CodeAnalysis.VisualBasic.VisualBasicCompilation) class.

Querying symbols

In this tutorial, you look at the "Hello World" program again. This time, you query the symbols in the program to understand what types those symbols represent. You query for the types in a namespace, and learn to find the methods available on a type.

You can see the finished code for this sample in our GitHub repository.

> **Note**

The Syntax Tree types use inheritance to describe the different syntax elements that are valid at different locations in the program. Using these APIs often means casting properties or collection members to specific derived types. In the following examples, the assignment and the casts are separate statements, using explicitly typed variables. You can read the code to see the return types of the API and the runtime type of the objects returned. In practice, it's more common to use implicitly typed variables and rely on API names to describe the type of objects being examined.

Create a new C# **Stand-Alone Code Analysis Tool** project:

- In Visual Studio, choose **File** > **New** > **Project** to display the New Project dialog.
- Under **Visual C#** > **Extensibility**, choose **Stand-Alone Code Analysis Tool**.
- Name your project "**SemanticQuickStart**" and click OK.

You're going to analyze the basic "Hello World!" program shown earlier. Add the text for the Hello World program as a constant in your `Program` class:

C#

```
        const string programText =
@"using System;
using System.Collections.Generic;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

Next, add the following code to build the syntax tree for the code text in the `programText` constant. Add the following line to your `Main` method:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(programText);

CompilationUnitSyntax root = tree.GetCompilationUnitRoot();
```

Next, build a CSharpCompilation from the tree you already created. The "Hello World" sample relies on the String and Console types. You need to reference the assembly that declares those two types in your compilation. Add the following line to your `Main` method to create a compilation of your syntax tree, including the reference to the appropriate assembly:

C#

```
var compilation = CSharpCompilation.Create("HelloWorld")
    .AddReferences(MetadataReference.CreateFromFile(
        typeof(string).Assembly.Location))
    .AddSyntaxTrees(tree);
```

The CSharpCompilation.AddReferences method adds references to the compilation. The MetadataReference.CreateFromFile method loads an assembly as a reference.

Querying the semantic model

Once you have a Compilation you can ask it for a SemanticModel for any SyntaxTree contained in that Compilation. You can think of the semantic model as the source for all the information you would normally get from intellisense. A SemanticModel can answer questions like "What names are in scope at this location?", "What members are accessible from this method?", "What variables are used in this block of text?", and "What does this name/expression refer to?" Add this statement to create the semantic model:

C#

```
SemanticModel model = compilation.GetSemanticModel(tree);
```

Binding a name

The Compilation creates the SemanticModel from the SyntaxTree. After creating the model, you can query it to find the first `using` directive, and retrieve the symbol information for the `System` namespace. Add these two lines to your `Main` method to create the semantic model and retrieve the symbol for the first `using` directive:

C#

```
// Use the syntax tree to find "using System;"
UsingDirectiveSyntax usingSystem = root.Usings[0];
NameSyntax systemName = usingSystem.Name;
```

```
// Use the semantic model for symbol information:
SymbolInfo nameInfo = model.GetSymbolInfo(systemName);
```

The preceding code shows how to bind the name in the first `using` directive to retrieve a Microsoft.CodeAnalysis.SymbolInfo for the `System` namespace. The preceding code also illustrates that you use the **syntax model** to find the structure of the code; you use the **semantic model** to understand its meaning. The **syntax model** finds the string `System` in the `using` directive. The **semantic model** has all the information about the types defined in the `System` namespace.

From the SymbolInfo object you can obtain the Microsoft.CodeAnalysis.ISymbol using the SymbolInfo.Symbol property. This property returns the symbol this expression refers to. For expressions that don't refer to anything (such as numeric literals) this property is `null`. When the SymbolInfo.Symbol is not null, the ISymbol.Kind denotes the type of the symbol. In this example, the ISymbol.Kind property is a SymbolKind.Namespace. Add the following code to your `Main` method. It retrieves the symbol for the `System` namespace and then displays all the child namespaces declared in the `System` namespace:

C#

```
var systemSymbol = (INamespaceSymbol?)nameInfo.Symbol;
if (systemSymbol?.GetNamespaceMembers() is not null)
{
    foreach (INamespaceSymbol ns in systemSymbol?.GetNamespaceMembers()!)
    {
        Console.WriteLine(ns);
    }
}
```

Run the program and you should see the following output:

Output

```
System.Collections
System.Configuration
System.Deployment
System.Diagnostics
System.Globalization
System.IO
System.Numerics
System.Reflection
System.Resources
System.Runtime
System.Security
System.StubHelpers
System.Text
```

```
System.Threading
Press any key to continue . . .
```

> **Note**
>
> The output does not include every namespace that is a child namespace of the `System` namespace. It displays every namespace that is present in this compilation, which only references the assembly where `System.String` is declared. Any namespaces declared in other assemblies are not known to this compilation

Binding an expression

The preceding code shows how to find a symbol by binding to a name. There are other expressions in a C# program that can be bound that aren't names. To demonstrate this capability, let's access the binding to a simple string literal.

The "Hello World" program contains a Microsoft.CodeAnalysis.CSharp.Syntax.LiteralExpressionSyntax, the "Hello, World!" string displayed to the console.

You find the "Hello, World!" string by locating the single string literal in the program. Then, once you've located the syntax node, get the type info for that node from the semantic model. Add the following code to your `Main` method:

C#

```
// Use the syntax model to find the literal string:
LiteralExpressionSyntax helloWorldString = root.DescendantNodes()
.OfType<LiteralExpressionSyntax>()
.Single();

// Use the semantic model for type information:
TypeInfo literalInfo = model.GetTypeInfo(helloWorldString);
```

The Microsoft.CodeAnalysis.TypeInfo struct includes a TypeInfo.Type property that enables access to the semantic information about the type of the literal. In this example, that's the `string` type. Add a declaration that assigns this property to a local variable:

C#

```
var stringTypeSymbol = (INamedTypeSymbol?)literalInfo.Type;
```

To finish this tutorial, let's build a LINQ query that creates a sequence of all the public methods declared on the `string` type that return a `string`. This query gets complex, so let's build it line by line, then reconstruct it as a single query. The source for this query is the sequence of all members declared on the `string` type:

```
C#

var allMembers = stringTypeSymbol?.GetMembers();
```

That source sequence contains all members, including properties and fields, so filter it using the ImmutableArray<T>.OfType method to find elements that are Microsoft.CodeAnalysis.IMethodSymbol objects:

```
C#

var methods = allMembers?.OfType<IMethodSymbol>();
```

Next, add another filter to return only those methods that are public and return a `string`:

```
C#

var publicStringReturningMethods = methods?
    .Where(m => SymbolEqualityComparer.Default.Equals(m.ReturnType,
stringTypeSymbol) &&
    m.DeclaredAccessibility == Accessibility.Public);
```

Select only the name property, and only distinct names by removing any overloads:

```
C#

var distinctMethods = publicStringReturningMethods?.Select(m =>
m.Name).Distinct();
```

You can also build the full query using the LINQ query syntax, and then display all the method names in the console:

```
C#

foreach (string name in (from method in stringTypeSymbol?
                         .GetMembers().OfType<IMethodSymbol>()
                         where
SymbolEqualityComparer.Default.Equals(method.ReturnType, stringTypeSymbol)
&&
                         method.DeclaredAccessibility ==
```

```
Accessibility.Public
                        select method.Name).Distinct())
{
    Console.WriteLine(name);
}
```

Build and run the program. You should see the following output:

Output

```
Join
Substring
Trim
TrimStart
TrimEnd
Normalize
PadLeft
PadRight
ToLower
ToLowerInvariant
ToUpper
ToUpperInvariant
ToString
Insert
Replace
Remove
Format
Copy
Concat
Intern
IsInterned
Press any key to continue . . .
```

You've used the Semantic API to find and display information about the symbols that are part of this program.

Get started with syntax transformation

Article • 09/15/2021

This tutorial builds on concepts and techniques explored in the Get started with syntax analysis and Get started with semantic analysis quickstarts. If you haven't already, you should complete those quickstarts before beginning this one.

In this quickstart, you explore techniques for creating and transforming syntax trees. In combination with the techniques you learned in previous quickstarts, you create your first command-line refactoring!

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Check the box for **.NET Compiler Platform SDK**. You'll find it last under the optional components.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab

4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Immutability and the .NET compiler platform

Immutability is a fundamental tenet of the .NET compiler platform. Immutable data structures can't be changed after they're created. Immutable data structures can be safely shared and analyzed by multiple consumers simultaneously. There's no danger that one consumer affects another in unpredictable ways. Your analyzer doesn't need locks or other concurrency measures. This rule applies to syntax trees, compilations, symbols, semantic models, and every other data structure you encounter. Instead of modifying existing structures, APIs create new objects based on specified differences to the old ones. You apply this concept to syntax trees to create new trees using transformations.

Create and transform trees

You choose one of two strategies for syntax transformations. **Factory methods** are best used when you're searching for specific nodes to replace, or specific locations where you want to insert new code. **Rewriters** are best when you want to scan an entire project for code patterns that you want to replace.

Create nodes with factory methods

The first syntax transformation demonstrates the factory methods. You're going to replace a `using System.Collections;` statement with a `using System.Collections.Generic;` statement. This example demonstrates how you create Microsoft.CodeAnalysis.CSharp.CSharpSyntaxNode objects using the Microsoft.CodeAnalysis.CSharp.SyntaxFactory factory methods. For each kind of **node**, **token**, or **trivia**, there's a factory method that creates an instance of that type. You create syntax trees by composing nodes hierarchically in a bottom-up fashion. Then, you'll transform the existing program by replacing existing nodes with the new tree you've created.

Start Visual Studio, and create a new C# **Stand-Alone Code Analysis Tool** project. In Visual Studio, choose **File** > **New** > **Project** to display the New Project dialog. Under **Visual C#** > **Extensibility** choose a **Stand-Alone Code Analysis Tool**. This quickstart has

two example projects, so name the solution **SyntaxTransformationQuickStart**, and name the project **ConstructionCS**. Click **OK**.

This project uses the Microsoft.CodeAnalysis.CSharp.SyntaxFactory class methods to construct a Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax representing the `System.Collections.Generic` namespace.

Add the following `using` directive to the top of the `Program.cs`.

C#

```
using static Microsoft.CodeAnalysis.CSharp.SyntaxFactory;
using static System.Console;
```

You'll create **name syntax nodes** to build the tree that represents the `using System.Collections.Generic;` statement. NameSyntax is the base class for four types of names that appear in C#. You compose these four types of names together to create any name that can appear in the C# language:

- Microsoft.CodeAnalysis.CSharp.Syntax.NameSyntax, which represents simple single identifier names like `System` and `Microsoft`.
- Microsoft.CodeAnalysis.CSharp.Syntax.GenericNameSyntax, which represents a generic type or method name such as `List<int>`.
- Microsoft.CodeAnalysis.CSharp.Syntax.QualifiedNameSyntax, which represents a qualified name of the form `<left-name>.<right-identifier-or-generic-name>` such as `System.IO`.
- Microsoft.CodeAnalysis.CSharp.Syntax.AliasQualifiedNameSyntax, which represents a name using an assembly extern alias such as `LibraryV2::Foo`.

You use the IdentifierName(String) method to create a NameSyntax node. Add the following code in your `Main` method in `Program.cs`:

C#

```
NameSyntax name = IdentifierName("System");
WriteLine($"\tCreated the identifier {name}");
```

The preceding code creates an IdentifierNameSyntax object and assigns it to the variable `name`. Many of the Roslyn APIs return base classes to make it easier to work with related types. The variable `name`, an NameSyntax, can be reused as you build the QualifiedNameSyntax. Don't use type inference as you build the sample. You'll automate that step in this project.

You've created the name. Now, it's time to build more nodes into the tree by building a QualifiedNameSyntax. The new tree uses `name` as the left of the name, and a new IdentifierNameSyntax for the `Collections` namespace as the right side of the QualifiedNameSyntax. Add the following code to `program.cs`:

```
C#
```

```
name = QualifiedName(name, IdentifierName("Collections"));
WriteLine(name.ToString());
```

Run the code again, and see the results. You're building a tree of nodes that represents code. You'll continue this pattern to build the QualifiedNameSyntax for the namespace `System.Collections.Generic`. Add the following code to `Program.cs`:

```
C#
```

```
name = QualifiedName(name, IdentifierName("Generic"));
WriteLine(name.ToString());
```

Run the program again to see that you've built the tree for the code to add.

Create a modified tree

You've built a small syntax tree that contains one statement. The APIs to create new nodes are the right choice to create single statements or other small code blocks. However, to build larger blocks of code, you should use methods that replace nodes or insert nodes into an existing tree. Remember that syntax trees are immutable. The **Syntax API** doesn't provide any mechanism for modifying an existing syntax tree after construction. Instead, it provides methods that produce new trees based on changes to existing ones. `With*` methods are defined in concrete classes that derive from SyntaxNode or in extension methods declared in the SyntaxNodeExtensions class. These methods create a new node by applying changes to an existing node's child properties. Additionally, the ReplaceNode extension method can be used to replace a descendent node in a subtree. This method also updates the parent to point to the newly created child and repeats this process up the entire tree - a process known as *re-spinning* the tree.

The next step is to create a tree that represents an entire (small) program and then modify it. Add the following code to the beginning of the `Program` class:

```
C#
```

```
        private const string sampleCode =
@"using System;
using System.Collections;
using System.Linq;
using System.Text;

namespace HelloWorld
{
    class Program
    {
        static void Main(string[] args)
        {
            Console.WriteLine(""Hello, World!"");
        }
    }
}";
```

> **Note**
>
> The example code uses the `System.Collections` namespace and not the `System.Collections.Generic` namespace.

Next, add the following code to the bottom of the `Main` method to parse the text and create a tree:

C#

```
SyntaxTree tree = CSharpSyntaxTree.ParseText(sampleCode);
var root = (CompilationUnitSyntax)tree.GetRoot();
```

This example uses the WithName(NameSyntax) method to replace the name in a UsingDirectiveSyntax node with the one constructed in the preceding code.

Create a new UsingDirectiveSyntax node using the WithName(NameSyntax) method to update the `System.Collections` name with the name you created in the preceding code. Add the following code to the bottom of the `Main` method:

C#

```
var oldUsing = root.Usings[1];
var newUsing = oldUsing.WithName(name);
WriteLine(root.ToString());
```

Run the program and look carefully at the output. The `newUsing` hasn't been placed in the root tree. The original tree hasn't been changed.

Add the following code using the ReplaceNode extension method to create a new tree. The new tree is the result of replacing the existing import with the updated `newUsing` node. You assign this new tree to the existing `root` variable:

```
C#

root = root.ReplaceNode(oldUsing, newUsing);
WriteLine(root.ToString());
```

Run the program again. This time the tree now correctly imports the `System.Collections.Generic` namespace.

Transform trees using `SyntaxRewriters`

The `With*` and ReplaceNode methods provide convenient means to transform individual branches of a syntax tree. The Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter class performs multiple transformations on a syntax tree. The Microsoft.CodeAnalysis.CSharp.CSharpSyntaxRewriter class is a subclass of Microsoft.CodeAnalysis.CSharp.CSharpSyntaxVisitor<TResult>. The CSharpSyntaxRewriter applies a transformation to a specific type of SyntaxNode. You can apply transformations to multiple types of SyntaxNode objects wherever they appear in a syntax tree. The second project in this quickstart creates a command-line refactoring that removes explicit types in local variable declarations anywhere that type inference could be used.

Create a new C# **Stand-Alone Code Analysis Tool** project. In Visual Studio, right-click the `SyntaxTransformationQuickStart` solution node. Choose **Add** > **New Project** to display the **New Project dialog**. Under **Visual C#** > **Extensibility**, choose **Stand-Alone Code Analysis Tool**. Name your project `TransformationCS` and click OK.

The first step is to create a class that derives from CSharpSyntaxRewriter to perform your transformations. Add a new class file to the project. In Visual Studio, choose **Project** > **Add Class...**. In the **Add New Item** dialog type `TypeInferenceRewriter.cs` as the filename.

Add the following `using` directives to the `TypeInferenceRewriter.cs` file:

```
C#

using Microsoft.CodeAnalysis;
using Microsoft.CodeAnalysis.CSharp;
```

```
using Microsoft.CodeAnalysis.CSharp.Syntax;
```

Next, make the `TypeInferenceRewriter` class extend the CSharpSyntaxRewriter class:

C#

```
public class TypeInferenceRewriter : CSharpSyntaxRewriter
```

Add the following code to declare a private read-only field to hold a SemanticModel and initialize it in the constructor. You will need this field later on to determine where type inference can be used:

C#

```
private readonly SemanticModel SemanticModel;

public TypeInferenceRewriter(SemanticModel semanticModel) => SemanticModel =
semanticModel;
```

Override the VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) method:

C#

```
public override SyntaxNode
VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax node)
{

}
```

> ⓘ **Note**
>
> Many of the Roslyn APIs declare return types that are base classes of the actual runtime types returned. In many scenarios, one kind of node may be replaced by another kind of node entirely - or even removed. In this example, the VisitLocalDeclarationStatement(LocalDeclarationStatementSyntax) method returns a SyntaxNode, instead of the derived type of LocalDeclarationStatementSyntax. This rewriter returns a new LocalDeclarationStatementSyntax node based on the existing one.

This quickstart handles local variable declarations. You could extend it to other declarations such as `foreach` loops, `for` loops, LINQ expressions, and lambda expressions. Furthermore this rewriter will only transform declarations of the simplest form:

C#

```
Type variable = expression;
```

If you want to explore on your own, consider extending the finished sample for these types of variable declarations:

C#

```
// Multiple variables in a single declaration.
Type variable1 = expression1,
     variable2 = expression2;
// No initializer.
Type variable;
```

Add the following code to the body of the `VisitLocalDeclarationStatement` method to skip rewriting these forms of declarations:

C#

```
if (node.Declaration.Variables.Count > 1)
{
    return node;
}
if (node.Declaration.Variables[0].Initializer == null)
{
    return node;
}
```

The method indicates that no rewriting takes place by returning the `node` parameter unmodified. If neither of those `if` expressions are true, the node represents a possible declaration with initialization. Add these statements to extract the type name specified in the declaration and bind it using the SemanticModel field to obtain a type symbol:

C#

```
var declarator = node.Declaration.Variables.First();
var variableTypeName = node.Declaration.Type;

var variableType = (ITypeSymbol)SemanticModel
    .GetSymbolInfo(variableTypeName)
    .Symbol;
```

Now, add this statement to bind the initializer expression:

C#

```
var initializerInfo =
SemanticModel.GetTypeInfo(declarator.Initializer.Value);
```

Finally, add the following `if` statement to replace the existing type name with the `var` keyword if the type of the initializer expression matches the type specified:

C#

```
if (SymbolEqualityComparer.Default.Equals(variableType,
initializerInfo.Type))
{
    TypeSyntax varTypeName = SyntaxFactory.IdentifierName("var")
        .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
        .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

    return node.ReplaceNode(variableTypeName, varTypeName);
}
else
{
    return node;
}
```

The conditional is required because the declaration may cast the initializer expression to a base class or interface. If that's desired, the types on the left and right-hand side of the assignment don't match. Removing the explicit type in these cases would change the semantics of a program. `var` is specified as an identifier rather than a keyword because `var` is a contextual keyword. The leading and trailing trivia (white space) are transferred from the old type name to the `var` keyword to maintain vertical white space and indentation. It's simpler to use `ReplaceNode` rather than `With*` to transform the LocalDeclarationStatementSyntax because the type name is actually the grandchild of the declaration statement.

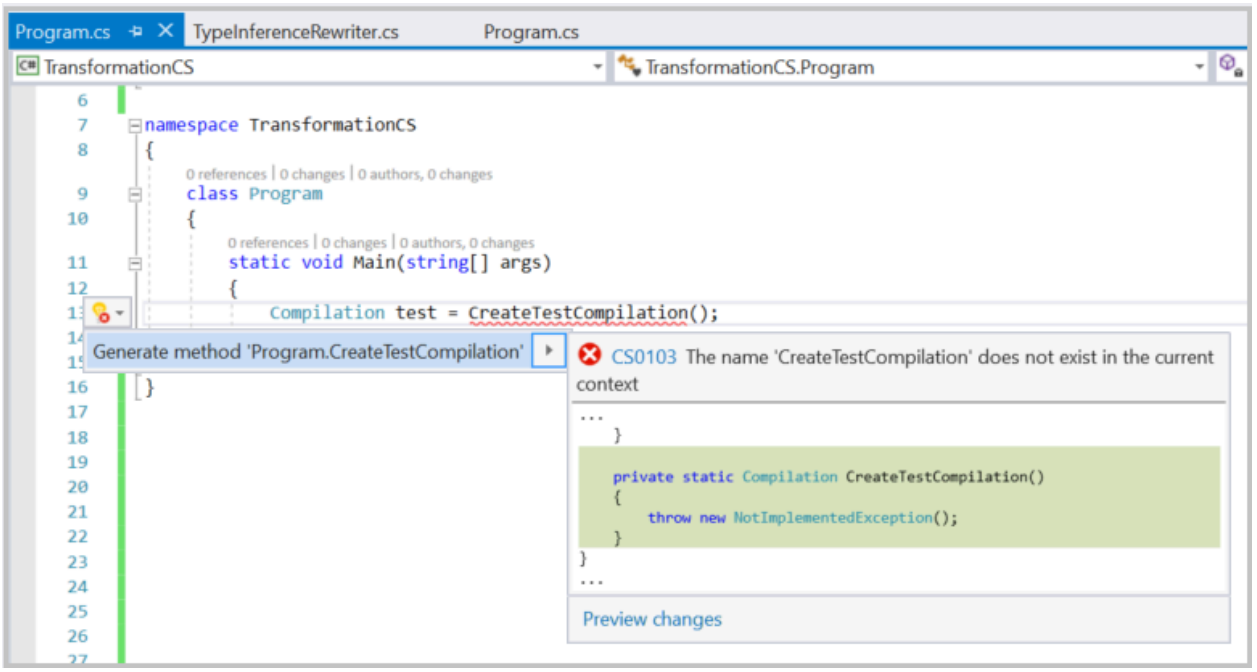
You've finished the `TypeInferenceRewriter`. Now return to your `Program.cs` file to finish the example. Create a test Compilation and obtain the SemanticModel from it. Use that SemanticModel to try your `TypeInferenceRewriter`. You'll do this step last. In the meantime declare a placeholder variable representing your test compilation:

C#

```
Compilation test = CreateTestCompilation();
```

After pausing a moment, you should see an error squiggle appear reporting that no `CreateTestCompilation` method exists. Press **Ctrl+Period** to open the light-bulb and then press Enter to invoke the **Generate Method Stub** command. This command will

generate a method stub for the `CreateTestCompilation` method in the `Program` class. You'll come back to fill in this method later:



Write the following code to iterate over each SyntaxTree in the test Compilation. For each one, initialize a new `TypeInferenceRewriter` with the SemanticModel for that tree:

C#

```
foreach (SyntaxTree sourceTree in test.SyntaxTrees)
{
    SemanticModel model = test.GetSemanticModel(sourceTree);

    TypeInferenceRewriter rewriter = new TypeInferenceRewriter(model);

    SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

    if (newSource != sourceTree.GetRoot())
    {
        File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
    }
}
```

Inside the `foreach` statement you created, add the following code to perform the transformation on each source tree. This code conditionally writes out the new transformed tree if any edits were made. Your rewriter should only modify a tree if it encounters one or more local variable declarations that could be simplified using type inference:

C#

```
SyntaxNode newSource = rewriter.Visit(sourceTree.GetRoot());

if (newSource != sourceTree.GetRoot())
{
    File.WriteAllText(sourceTree.FilePath, newSource.ToFullString());
}
```

You should see squiggles under the `File.WriteAllText` code. Select the light bulb, and add the necessary `using System.IO;` statement.

You're almost done! There's one step left: creating a test Compilation. Since you haven't been using type inference at all during this quickstart, it would have made a perfect test case. Unfortunately, creating a Compilation from a C# project file is beyond the scope of this walkthrough. But fortunately, if you've been following instructions carefully, there's hope. Replace the contents of the `CreateTestCompilation` method with the following code. It creates a test compilation that coincidentally matches the project described in this quickstart:

C#

```
String programPath = @"..\..\..\Program.cs";
String programText = File.ReadAllText(programPath);
SyntaxTree programTree =
               CSharpSyntaxTree.ParseText(programText)
                               .WithFilePath(programPath);

String rewriterPath = @"..\..\..\TypeInferenceRewriter.cs";
String rewriterText = File.ReadAllText(rewriterPath);
SyntaxTree rewriterTree =
               CSharpSyntaxTree.ParseText(rewriterText)
                               .WithFilePath(rewriterPath);

SyntaxTree[] sourceTrees = { programTree, rewriterTree };

MetadataReference mscorlib =
        MetadataReference.CreateFromFile(typeof(object).Assembly.Location);
MetadataReference codeAnalysis =

MetadataReference.CreateFromFile(typeof(SyntaxTree).Assembly.Location);
MetadataReference csharpCodeAnalysis =

MetadataReference.CreateFromFile(typeof(CSharpSyntaxTree).Assembly.Location)
;

MetadataReference[] references = { mscorlib, codeAnalysis,
csharpCodeAnalysis };

return CSharpCompilation.Create("TransformationCS",
    sourceTrees,
```

```
    references,
    new CSharpCompilationOptions(OutputKind.ConsoleApplication));
```

Cross your fingers and run the project. In Visual Studio, choose **Debug** > **Start Debugging**. You should be prompted by Visual Studio that the files in your project have changed. Click "**Yes to All**" to reload the modified files. Examine them to observe your awesomeness. Note how much cleaner the code looks without all those explicit and redundant type specifiers.

Congratulations! You've used the **Compiler APIs** to write your own refactoring that searches all files in a C# project for certain syntactic patterns, analyzes the semantics of source code that matches those patterns, and transforms it. You're now officially a refactoring author!

Tutorial: Write your first analyzer and code fix

Summarize this article for me

The .NET Compiler Platform SDK provides the tools you need to create custom diagnostics (analyzers), code fixes, code refactoring, and diagnostic suppressors that target C# or Visual Basic code. An **analyzer** contains code that recognizes violations of your rule. Your **code fix** contains the code that fixes the violation. The rules you implement can be anything from code structure to coding style to naming conventions and more. The .NET Compiler Platform provides the framework for running analysis as developers are writing code, and all the Visual Studio UI features for fixing code: showing squiggles in the editor, populating the Visual Studio Error List, creating the "light bulb" suggestions, and showing the rich preview of the suggested fixes.

In this tutorial, you explore the creation of an **analyzer** and an accompanying **code fix** using the Roslyn APIs. An analyzer is a way to perform source code analysis and report a problem to the user. Optionally, you can associate a code fix with the analyzer to represent a modification to the user's source code. This tutorial creates an analyzer that finds local variable declarations that could be declared using the `const` modifier but aren't. The accompanying code fix modifies those declarations to add the `const` modifier.

Prerequisites

- Visual Studio

You need to install the **.NET Compiler Platform SDK** by using the Visual Studio Installer:

Installation instructions - Visual Studio Installer

There are two different ways to find the **.NET Compiler Platform SDK** in the **Visual Studio Installer**:

Install using the Visual Studio Installer - Workloads view

The .NET Compiler Platform SDK is not automatically selected as part of the Visual Studio extension development workload. You must select it as an optional component.

1. Run **Visual Studio Installer**
2. Select **Modify**

3. Check the **Visual Studio extension development** workload.
4. Open the **Visual Studio extension development** node in the summary tree.
5. Make sure the box for **.NET Compiler Platform SDK** is checked.
6. Select **Modify**.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Open the **Individual components** node in the summary tree.
2. Check the box for **DGML editor**

Install using the Visual Studio Installer - Individual components tab

1. Run **Visual Studio Installer**
2. Select **Modify**
3. Select the **Individual components** tab
4. Check the box for **.NET Compiler Platform SDK**. You'll find it at the top under the **Compilers, build tools, and runtimes** section.
5. Select **Modify**.

Optionally, you'll also want the **DGML editor** to display graphs in the visualizer:

1. Check the box for **DGML editor**. You'll find it under the **Code tools** section.

Creating and validating your analyzer involves several steps:

1. Create the solution.
2. Register the analyzer name and description.
3. Report analyzer warnings and recommendations.
4. Implement the code fix to accept recommendations.
5. Improve the analysis through unit tests.

Create the solution

1. In Visual Studio, select **File** > **New** > **Project/Solution** to open the New Project dialog.
2. In the search box, type **Analyzer** to find the **Analyzer with Code Fix (.NET Standard)** template.
3. Select **Next**.
4. Name your project "**MakeConst**" and select **Create**.

> ⓘ **Note**

You might get a compilation error (*MSB4062: The "CompareBuildTaskVersion" task couldn't be loaded*). To fix this error, update the NuGet packages in the solution by using NuGet Package Manager or use `Update-Package` in the Package Manager Console window.

Explore the analyzer template

The analyzer with code fix template creates five projects:

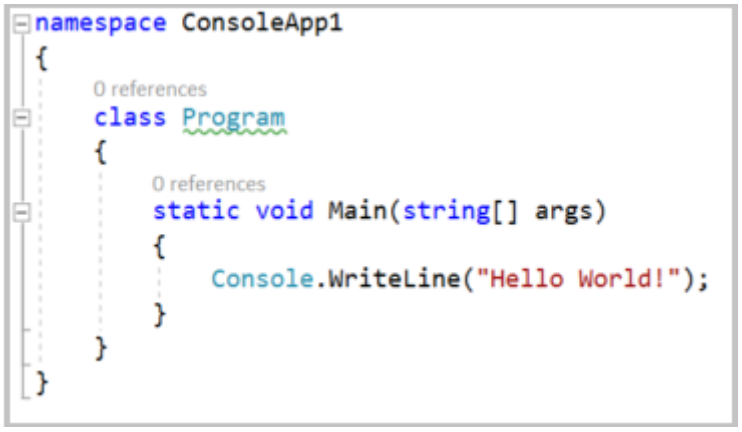
- **MakeConst**, which contains the analyzer.
- **MakeConst.CodeFixes**, which contains the code fix.
- **MakeConst.Package**, which is used to produce NuGet package for the analyzer and code fix.
- **MakeConst.Test**, which is a unit test project.
- **MakeConst.Vsix**, which is the default startup project that starts a second instance of Visual Studio that has loaded your new analyzer. Press `F5` to start the VSIX project.

> **Note**
>
> Analyzers should target .NET Standard 2.0 because they can run in .NET Core environment (command line builds) and .NET Framework environment (Visual Studio).

> **Tip**
>
> When you run your analyzer, you start a second copy of Visual Studio. This second copy uses a different registry hive to store settings. That difference enables you to differentiate the visual settings in the two copies of Visual Studio. You can pick a different theme for the experimental run of Visual Studio. In addition, don't roam your settings or sign in to your Visual Studio account by using the experimental run of Visual Studio. That keeps the settings different.
>
> The hive includes not only the analyzer under development, but also any previous analyzers opened. To reset Roslyn hive, you need to manually delete it from *%LocalAppData%\Microsoft\VisualStudio*. The folder name of Roslyn hive ends in `Roslyn`, for example, `16.0_9ae182f9Roslyn`. You might need to clean the solution and rebuild it after deleting the hive.

In the second Visual Studio instance that you just started, create a new C# Console Application project (any target framework will work -- analyzers work at the source level.) Hover over the token with a wavy underline, and the warning text provided by an analyzer appears.

The template creates an analyzer that reports a warning on each type declaration where the type name contains lowercase letters, as shown in the following figure:



The template also provides a code fix that changes any type name containing lower case characters to all upper case. You can select the light bulb displayed with the warning to see the suggested changes. Accepting the suggested changes updates the type name and all references to that type in the solution. Now that you see the initial analyzer in action, close the second Visual Studio instance and return to your analyzer project.

You don't have to start a second copy of Visual Studio and create new code to test every change in your analyzer. The template also creates a unit test project for you. That project contains two tests. `TestMethod1` shows the typical format of a test that analyzes code without triggering a diagnostic. `TestMethod2` shows the format of a test that triggers a diagnostic, and then applies a suggested code fix. As you build your analyzer and code fix, you'll write tests for different code structures to verify your work. Unit tests for analyzers are much quicker than testing them interactively with Visual Studio.

> **Tip**
>
> Analyzer unit tests are a great tool when you know what code constructs should and shouldn't trigger your analyzer. Loading your analyzer in another copy of Visual Studio is a great tool to explore and find constructs you might not have thought about yet.

In this tutorial, you write an analyzer that reports to the user any local variable declarations that can be converted to local constants. For example, consider the following code:

C#

```
int x = 0;
Console.WriteLine(x);
```

In the code preceding, `x` is assigned a constant value and is never modified. It can be declared by using the `const` modifier:

C#

```
const int x = 0;
Console.WriteLine(x);
```

The analysis to determine whether a variable can be made constant is involved, requiring syntactic analysis, constant analysis of the initializer expression, and dataflow analysis to ensure that the variable is never written to. The .NET Compiler Platform provides APIs that make it easier to perform this analysis.

Create analyzer registrations

The template creates the initial `DiagnosticAnalyzer` class in the *MakeConstAnalyzer.cs* file. This initial analyzer shows two important properties of every analyzer.

- Every diagnostic analyzer must provide a `[DiagnosticAnalyzer]` attribute that describes the language it operates on.
- Every diagnostic analyzer must derive (directly or indirectly) from the DiagnosticAnalyzer class.

The template also shows the basic features that are part of any analyzer:

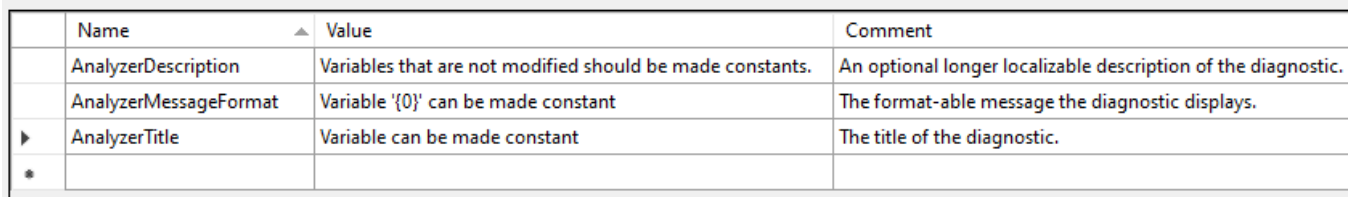
1. Register actions. The actions represent code changes that should trigger your analyzer to examine code for violations. When Visual Studio detects code edits that match a registered action, it calls your analyzer's registered method.
2. Create diagnostics. When your analyzer detects a violation, it creates a diagnostic object that Visual Studio uses to notify the user of the violation.

You register actions in your override of DiagnosticAnalyzer.Initialize(AnalysisContext) method. In this tutorial, you'll visit **syntax nodes** looking for local declarations, and see which of those have constant values. If a declaration could be constant, your analyzer creates and reports a diagnostic.

The first step is to update the registration constants and `Initialize` method so these constants indicate your "Make Const" analyzer. Most of the string constants are defined in the string resource file. Follow that practice for easier localization. Open the *Resources.resx* file for the **MakeConst** analyzer project. This action displays the resource editor. Update the string resources as follows:

- Change `AnalyzerDescription` to "Variables that aren't modified should be made constants.".
- Change `AnalyzerMessageFormat` to "Variable '{0}' can be made constant".
- Change `AnalyzerTitle` to "Variable can be made constant".

When you finish, the resource editor should appear as shown in the following figure:



	Name	Value	Comment
	AnalyzerDescription	Variables that are not modified should be made constants.	An optional longer localizable description of the diagnostic.
	AnalyzerMessageFormat	Variable '{0}' can be made constant	The format-able message the diagnostic displays.
▶	AnalyzerTitle	Variable can be made constant	The title of the diagnostic.
*			

Make the remaining changes in the analyzer file. Open *MakeConstAnalyzer.cs* in Visual Studio. Change the registered action from one that acts on symbols to one that acts on syntax. In the `MakeConstAnalyzerAnalyzer.Initialize` method, find the line that registers the action on symbols:

C#

```
context.RegisterSymbolAction(AnalyzeSymbol, SymbolKind.NamedType);
```

Replace it with the following line:

C#

```
context.RegisterSyntaxNodeAction(AnalyzeNode,
SyntaxKind.LocalDeclarationStatement);
```

After that change, you can delete the `AnalyzeSymbol` method. This analyzer examines SyntaxKind.LocalDeclarationStatement, not SymbolKind.NamedType statements. Notice that `AnalyzeNode` has red squiggles under it. The code you just added references an `AnalyzeNode` method that isn't declared. Declare that method by using the following code:

C#

```
private void AnalyzeNode(SyntaxNodeAnalysisContext context)
{
}
```

Change the `Category` to "Usage" in *MakeConstAnalyzer.cs* as shown in the following code:

C#

```
private const string Category = "Usage";
```

Find local declarations that could be const

Now you're ready to write the first version of the `AnalyzeNode` method. It should look for a single local declaration that could be `const` but isn't, like the following code:

C#

```
int x = 0;
Console.WriteLine(x);
```

The first step is to find local declarations. Add the following code to `AnalyzeNode` in *MakeConstAnalyzer.cs*:

C#

```
var localDeclaration = (LocalDeclarationStatementSyntax)context.Node;
```

This cast always succeeds because your analyzer registered for changes to local declarations, and only local declarations. No other node type triggers a call to your `AnalyzeNode` method. Next, check the declaration for any `const` modifiers. If you find them, return immediately. The following code looks for any `const` modifiers on the local declaration:

C#

```
// make sure the declaration isn't already const:
if (localDeclaration.Modifiers.Any(SyntaxKind.ConstKeyword))
{
    return;
}
```

Finally, you need to check that the variable could be `const`. That check makes sure the variable is never assigned after it's initialized.

Perform some semantic analysis by using the SyntaxNodeAnalysisContext. Use the `context` argument to determine whether the local variable declaration can be made `const`. A Microsoft.CodeAnalysis.SemanticModel represents all of semantic information in a single source file. You can learn more in the article that covers semantic models. Use the Microsoft.CodeAnalysis.SemanticModel to perform data flow analysis on the local declaration statement. Then, use the results of this data flow analysis to ensure that the local variable isn't written with a new value anywhere else. Call the GetDeclaredSymbol extension member to

retrieve the ILocalSymbol for the variable and check that it isn't contained within the DataFlowAnalysis.WrittenOutside collection of the data flow analysis. Add the following code to the end of the `AnalyzeNode` method:

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

The code you just added ensures that the variable isn't modified, and can therefore be made `const`. It's time to raise the diagnostic. Add the following code as the last line in `AnalyzeNode`:

C#

```
context.ReportDiagnostic(Diagnostic.Create(Rule, context.Node.GetLocation(),
localDeclaration.Declaration.Variables.First().Identifier.ValueText));
```

You can check your progress by pressing `F5` to run your analyzer. You can load the console application you created earlier and then add the following test code:

C#

```
int x = 0;
Console.WriteLine(x);
```

The light bulb should appear, and your analyzer should report a diagnostic. However, depending on your version of Visual Studio, you'll either see:

- The light bulb, which still uses the template generated code fix, will tell you it can be made upper case.
- A banner message at the top of the editor saying the 'MakeConstCodeFixProvider' encountered an error and has been disabled. This error occurs because the code fix provider hasn't yet been changed and still expects to find `TypeDeclarationSyntax` elements instead of `LocalDeclarationStatementSyntax` elements.

The next section explains how to write the code fix.

Write the code fix

An analyzer can provide one or more code fixes. A code fix defines an edit that addresses the reported issue. For the analyzer that you created, you can provide a code fix that inserts the const keyword:

diff

```
- int x = 0;
+ const int x = 0;
Console.WriteLine(x);
```

The user chooses it from the light bulb UI in the editor and Visual Studio changes the code.

Open *CodeFixResources.resx* file and change `CodeFixTitle` to "Make constant".

Open the *MakeConstCodeFixProvider.cs* file added by the template. This code fix is already wired up to the Diagnostic ID produced by your diagnostic analyzer, but it doesn't yet implement the right code transform.

Next, delete the `MakeUppercaseAsync` method. It no longer applies.

All code fix providers derive from CodeFixProvider. They all override CodeFixProvider.RegisterCodeFixesAsync(CodeFixContext) to report available code fixes. In `RegisterCodeFixesAsync`, change the ancestor node type you're searching for to a LocalDeclarationStatementSyntax to match the diagnostic:

C#

```
var declaration =
root.FindToken(diagnosticSpan.Start).Parent.AncestorsAndSelf().OfType<LocalDeclarat
ionStatementSyntax>().First();
```

Next, change the last line to register a code fix. Your fix creates a new document that results from adding the `const` modifier to an existing declaration:

C#

```
// Register a code action that will invoke the fix.
context.RegisterCodeFix(
    CodeAction.Create(
        title: CodeFixResources.CodeFixTitle,
        createChangedDocument: c => MakeConstAsync(context.Document, declaration,
```

```
c),
        equivalenceKey: nameof(CodeFixResources.CodeFixTitle)),
    diagnostic);
```

You'll notice red squiggles in the code you just added on the symbol `MakeConstAsync`. Add a declaration for `MakeConstAsync` like the following code:

C#

```
private static async Task<Document> MakeConstAsync(Document document,
    LocalDeclarationStatementSyntax localDeclaration,
    CancellationToken cancellationToken)
{
}
```

Your new `MakeConstAsync` method transforms the Document representing the user's source file into a new Document that now contains a `const` declaration.

You create a new `const` keyword token to insert at the front of the declaration statement. Be careful to first remove any leading trivia from the first token of the declaration statement and attach it to the `const` token. Add the following code to the `MakeConstAsync` method:

C#

```
// Remove the leading trivia from the local declaration.
SyntaxToken firstToken = localDeclaration.GetFirstToken();
SyntaxTriviaList leadingTrivia = firstToken.LeadingTrivia;
LocalDeclarationStatementSyntax trimmedLocal = localDeclaration.ReplaceToken(
    firstToken, firstToken.WithLeadingTrivia(SyntaxTriviaList.Empty));

// Create a const token with the leading trivia.
SyntaxToken constToken = SyntaxFactory.Token(leadingTrivia,
SyntaxKind.ConstKeyword, SyntaxFactory.TriviaList(SyntaxFactory.ElasticMarker));
```

Next, add the `const` token to the declaration using the following code:

C#

```
// Insert the const token into the modifiers list, creating a new modifiers list.
SyntaxTokenList newModifiers = trimmedLocal.Modifiers.Insert(0, constToken);
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal
    .WithModifiers(newModifiers)
    .WithDeclaration(localDeclaration.Declaration);
```

Next, format the new declaration to match C# formatting rules. Formatting your changes to match existing code creates a better experience. Add the following statement immediately

after the existing code:

```C#
// Add an annotation to format the new local declaration.
LocalDeclarationStatementSyntax formattedLocal =
newLocal.WithAdditionalAnnotations(Formatter.Annotation);
```

A new namespace is required for this code. Add the following `using` directive to the top of the file:

```C#
using Microsoft.CodeAnalysis.Formatting;
```

The final step is to make your edit. There are three steps to this process:

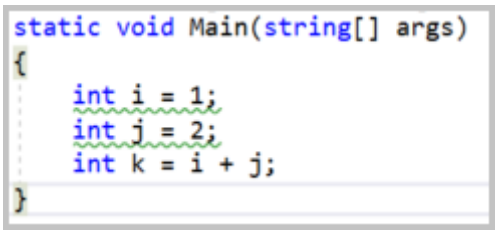
1. Get a handle to the existing document.
2. Create a new document by replacing the existing declaration with the new declaration.
3. Return the new document.

Add the following code to the end of the `MakeConstAsync` method:

```C#
// Replace the old local declaration with the new local declaration.
SyntaxNode oldRoot = await
document.GetSyntaxRootAsync(cancellationToken).ConfigureAwait(false);
SyntaxNode newRoot = oldRoot.ReplaceNode(localDeclaration, formattedLocal);

// Return document with transformed tree.
return document.WithSyntaxRoot(newRoot);
```

Your code fix is ready to try. Press `F5` to run the analyzer project in a second instance of Visual Studio. In the second Visual Studio instance, create a new C# Console Application project and add a few local variable declarations initialized with constant values to the Main method. You see that they're reported as warnings as shown in the following image.



You've made a lot of progress. There are squiggles under the declarations that can be made `const`. But there is still work to do. This works fine if you add `const` to the declarations starting

with `i`, then `j`, and finally `k`. But, if you add the `const` modifier in a different order, starting with `k`, your analyzer creates errors: `k` can't be declared `const`, unless `i` and `j` are both already `const`. You need to do more analysis to ensure you handle the different ways variables can be declared and initialized.

Build unit tests

Your analyzer and code fix work on a simple case of a single declaration that you can make const. Numerous possible declaration statements exist where this implementation makes mistakes. You'll address these cases by working with the unit test library written by the template. It's much faster than repeatedly opening a second copy of Visual Studio.

Open the *MakeConstUnitTests.cs* file in the unit test project. The template created two tests that follow the two common patterns for an analyzer and code fix unit test. `TestMethod1` shows the pattern for a test that ensures the analyzer doesn't report a diagnostic when it shouldn't. `TestMethod2` shows the pattern for reporting a diagnostic and running the code fix.

The template uses Microsoft.CodeAnalysis.Testing packages for unit testing.

> **Tip**
>
> The testing library supports a special markup syntax, including the following:
>
> - `[|text|]`: indicates that a diagnostic is reported for `text`. By default, this form may only be used for testing analyzers with exactly one `DiagnosticDescriptor` provided by `DiagnosticAnalyzer.SupportedDiagnostics`.
> - `{|ExpectedDiagnosticId:text|}`: indicates that a diagnostic with **Id** `ExpectedDiagnosticId` is reported for `text`.

Replace the template tests in the `MakeConstUnitTest` class with the following test method:

C#

```
        [TestMethod]
        public async Task LocalIntCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
```

```
        [|int i = 0;|]
        Console.WriteLine(i);
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

Run this test to make sure it passes. In Visual Studio, open the **Test Explorer** by selecting **Test** > **Windows** > **Test Explorer**. Then select **Run All**.

Create tests for valid declarations

As a general rule, analyzers should exit as quickly as possible, doing minimal work. Visual Studio calls registered analyzers as the user edits code. Responsiveness is a key requirement. Several test cases exist for code that shouldn't raise your diagnostic. Your analyzer already handles several of those tests. Add the following test methods to represent those cases:

C#

```
        [TestMethod]
        public async Task VariableIsAssigned_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0;
        Console.WriteLine(i++);
    }
}
");
        }
```

C#

```
        [TestMethod]
        public async Task VariableIsAlreadyConst_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        const int i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

C#

```
        [TestMethod]
        public async Task NoInitializer_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i;
        i = 0;
        Console.WriteLine(i);
    }
}
");
        }
```

These tests pass because your analyzer already handles these conditions:

- Variables assigned after initialization are detected by data flow analysis.
- Declarations that are already `const` are filtered out by checking for the `const` keyword.
- Declarations with no initializer are handled by the data flow analysis that detects assignments outside the declaration.

Next, add test methods for conditions you haven't handled yet:

- Declarations where the initializer isn't a constant, because they can't be compile-time constants:

```
```

C#

```
        [TestMethod]
        public async Task InitializerIsNotConstant_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
    }
}
");
        }
```

It can be even more complicated because C# allows multiple declarations as one statement. Consider the following test case string constant:

C#

```
        [TestMethod]
        public async Task MultipleInitializers_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int i = 0, j = DateTime.Now.DayOfYear;
        Console.WriteLine(i);
        Console.WriteLine(j);
    }
}
");
        }
```

The variable `i` can be constant, but the variable `j` can't. Therefore, this statement can't be a const declaration.

Run your tests again, and you see these last two test cases fail.

Update your analyzer to ignore correct declarations

You need to enhance your analyzer's `AnalyzeNode` method to filter out code that matches these conditions. These conditions are all related, so similar changes fix all these conditions. Make the following changes to `AnalyzeNode`:

- Your semantic analysis examined a single variable declaration. This code needs to be in a `foreach` loop that examines all the variables declared in the same statement.
- Each declared variable needs to have an initializer.
- Each declared variable's initializer must be a compile-time constant.

In your `AnalyzeNode` method, replace the original semantic analysis:

C#

```
// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

// Retrieve the local symbol for each variable in the local declaration
// and ensure that it is not written outside of the data flow analysis region.
VariableDeclaratorSyntax variable =
localDeclaration.Declaration.Variables.Single();
ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
{
    return;
}
```

with the following code snippet:

C#

```
// Ensure that all variables in the local declaration have initializers that
// are assigned with constant values.
foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    EqualsValueClauseSyntax initializer = variable.Initializer;
    if (initializer == null)
    {
        return;
    }

    Optional<object> constantValue =
context.SemanticModel.GetConstantValue(initializer.Value,
context.CancellationToken);
    if (!constantValue.HasValue)
    {
        return;
    }
```

```
}

// Perform data flow analysis on the local declaration.
DataFlowAnalysis dataFlowAnalysis =
context.SemanticModel.AnalyzeDataFlow(localDeclaration);

foreach (VariableDeclaratorSyntax variable in
localDeclaration.Declaration.Variables)
{
    // Retrieve the local symbol for each variable in the local declaration
    // and ensure that it is not written outside of the data flow analysis region.
    ISymbol variableSymbol = context.SemanticModel.GetDeclaredSymbol(variable,
context.CancellationToken);
    if (dataFlowAnalysis.WrittenOutside.Contains(variableSymbol))
    {
        return;
    }
}
```

The first `foreach` loop examines each variable declaration by using syntactic analysis. The first check guarantees that the variable has an initializer. The second check guarantees that the initializer is a constant. The second loop has the original semantic analysis. The semantic checks are in a separate loop because they have a greater impact on performance. Run your tests again, and you should see them all pass.

Add the final polish

You're almost done. There are a few more conditions for your analyzer to handle. Visual Studio calls analyzers while the user is writing code. It's often the case that your analyzer is called for code that doesn't compile. The diagnostic analyzer's `AnalyzeNode` method doesn't check if the constant value is convertible to the variable type. So, the current implementation happily converts an incorrect declaration such as `int i = "abc"` to a local constant. Add a test method for this case:

C#

```
        [TestMethod]
        public async Task DeclarationIsInvalid_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        int x = {|CS0029:""abc""|};
    }
```

```
}
");
        }
```

In addition, reference types aren't handled properly. The only constant value allowed for a reference type is `null`, except in the case of System.String, which allows string literals. In other words, `const string s = "abc"` is legal, but `const object s = "abc"` is not. This code snippet verifies that condition:

C#

```
        [TestMethod]
        public async Task DeclarationIsNotString_NoDiagnostic()
        {
            await VerifyCS.VerifyAnalyzerAsync(@"
using System;

class Program
{
    static void Main()
    {
        object s = ""abc"";
    }
}
");
        }
```

To be thorough, you need to add another test to make sure that you can create a constant declaration for a string. The following snippet defines both the code that raises the diagnostic, and the code after the fix is applied:

C#

```
        [TestMethod]
        public async Task StringCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|string s = ""abc"";|]
    }
}
", @"
using System;

class Program
```

```
{
    static void Main()
    {
        const string s = ""abc"";
    }
}
");
        }
```

Finally, if a variable is declared with the `var` keyword, the code fix does the wrong thing and generates a `const var` declaration, which the C# language doesn't support. To fix this bug, the code fix must replace the `var` keyword with the inferred type's name:

C#

```
        [TestMethod]
        public async Task VarIntDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = 4;|]
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const int item = 4;
    }
}
");
        }

        [TestMethod]
        public async Task VarStringDeclarationCouldBeConstant_Diagnostic()
        {
            await VerifyCS.VerifyCodeFixAsync(@"
using System;

class Program
{
    static void Main()
    {
        [|var item = ""abc"";|]
```

```
    }
}
", @"
using System;

class Program
{
    static void Main()
    {
        const string item = ""abc"";
    }
}
");
        }
```

Fortunately, you can address all of the above bugs by using the same techniques that you just learned.

To fix the first bug, open *MakeConstAnalyzer.cs* and locate the `foreach` loop where each of the local declaration's initializers are checked to ensure that they're assigned constant values. Immediately *before* the first `foreach` loop, call `context.SemanticModel.GetTypeInfo()` to retrieve detailed information about the declared type of the local declaration:

C#

```
TypeSyntax variableTypeName = localDeclaration.Declaration.Type;
ITypeSymbol variableType = context.SemanticModel.GetTypeInfo(variableTypeName,
context.CancellationToken).ConvertedType;
```

Then, inside your `foreach` loop, check each initializer to make sure it's convertible to the variable type. Add the following check after ensuring that the initializer is a constant:

C#

```
// Ensure that the initializer value can be converted to the type of the
// local declaration without a user-defined conversion.
Conversion conversion = context.SemanticModel.ClassifyConversion(initializer.Value,
variableType);
if (!conversion.Exists || conversion.IsUserDefined)
{
    return;
}
```

The next change builds upon the last one. Before the closing curly brace of the first `foreach` loop, add the following code to check the type of the local declaration when the constant is a string or null.

```
```

C#

```
// Special cases:
//  * If the constant value is a string, the type of the local declaration
//    must be System.String.
//  * If the constant value is null, the type of the local declaration must
//    be a reference type.
if (constantValue.Value is string)
{
    if (variableType.SpecialType != SpecialType.System_String)
    {
        return;
    }
}
else if (variableType.IsReferenceType && constantValue.Value != null)
{
    return;
}
```

You must write a bit more code in your code fix provider to replace the `var` keyword with the correct type name. Return to *MakeConstCodeFixProvider.cs*. The code you'll add does the following steps:

- Check if the declaration is a `var` declaration, and if it is:
- Create a new type for the inferred type.
- Make sure the type declaration is not an alias. If so, it is legal to declare `const var`.
- Make sure that `var` isn't a type name in this program. (If so, `const var` is legal).
- Simplify the full type name

That sounds like a lot of code. It's not. Replace the line that declares and initializes `newLocal` with the following code. It goes immediately after the initialization of `newModifiers`:

C#

```
// If the type of the declaration is 'var', create a new type name
// for the inferred type.
VariableDeclarationSyntax variableDeclaration = localDeclaration.Declaration;
TypeSyntax variableTypeName = variableDeclaration.Type;
if (variableTypeName.IsVar)
{
    SemanticModel semanticModel = await
document.GetSemanticModelAsync(cancellationToken).ConfigureAwait(false);

    // Special case: Ensure that 'var' isn't actually an alias to another type
    // (e.g. using var = System.String).
    IAliasSymbol aliasInfo = semanticModel.GetAliasInfo(variableTypeName,
cancellationToken);
    if (aliasInfo == null)
    {
        // Retrieve the type inferred for var.
```

```
        ITypeSymbol type = semanticModel.GetTypeInfo(variableTypeName,
cancellationToken).ConvertedType;

        // Special case: Ensure that 'var' isn't actually a type named 'var'.
        if (type.Name != "var")
        {
            // Create a new TypeSyntax for the inferred type. Be careful
            // to keep any leading and trailing trivia from the var keyword.
            TypeSyntax typeName =
SyntaxFactory.ParseTypeName(type.ToDisplayString())
                .WithLeadingTrivia(variableTypeName.GetLeadingTrivia())
                .WithTrailingTrivia(variableTypeName.GetTrailingTrivia());

            // Add an annotation to simplify the type name.
            TypeSyntax simplifiedTypeName =
typeName.WithAdditionalAnnotations(Simplifier.Annotation);

            // Replace the type in the variable declaration.
            variableDeclaration = variableDeclaration.WithType(simplifiedTypeName);
        }
    }
}
// Produce the new local declaration.
LocalDeclarationStatementSyntax newLocal = trimmedLocal.WithModifiers(newModifiers)
                           .WithDeclaration(variableDeclaration);
```

You'll need to add one `using` directive to use the Simplifier type:

C#

```
using Microsoft.CodeAnalysis.Simplification;
```

Run your tests, and they should all pass. Congratulate yourself by running your finished analyzer. Press `Ctrl` + `F5` to run the analyzer project in a second instance of Visual Studio with the Roslyn Preview extension loaded.

- In the second Visual Studio instance, create a new C# Console Application project and add `int x = "abc";` to the Main method. Thanks to the first bug fix, no warning is reported for this local variable declaration (though there's a compiler error as expected).
- Next, add `object s = "abc";` to the Main method. Because of the second bug fix, no warning is reported.
- Finally, add another local variable that uses the `var` keyword. You see that a warning is reported and a suggestion appears beneath to the left.
- Move the editor caret over the squiggly underline and press `Ctrl` + `.`. to display the suggested code fix. Upon selecting your code fix, note that the `var` keyword is now handled correctly.

Finally, add the following code:

C#

```
int i = 2;
int j = 32;
int k = i + j;
```

After these changes, you get red squiggles only on the first two variables. Add `const` to both `i` and `j`, and you get a new warning on `k` because it can now be `const`.

Congratulations! You created your first .NET Compiler Platform extension that performs on-the-fly code analysis to detect an issue and provides a quick fix to correct it. Along the way, you learned many of the code APIs that are part of the .NET Compiler Platform SDK (Roslyn APIs). You can check your work against the completed sample in the samples GitHub repository.

Other resources

- Get started with syntax analysis
- Get started with semantic analysis

Last updated on 02/17/2026

Programming concepts (C#)

10/10/2025

This section explains programming concepts in the C# language.

In this section

Expand table

Title	Description
Covariance and Contravariance (C#)	Shows how to enable implicit conversion of generic type parameters in interfaces and delegates.
Iterators (C#)	Describes iterators, which are used to step through collections and return elements one at a time.

Related sections

- Performance Tips

  Discusses several basic rules that might help you increase the performance of your application.

Covariance and Contravariance (C#)

Article • 07/30/2022

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.

C#

```
// 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.

C#

```
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.

```
C#
```

```
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 and later versions, 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.

```
C#
```

```
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

Expand table

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

Variance in Generic Interfaces (C#)

Article • 09/15/2021

.NET Framework 4 introduced variance support for several existing generic interfaces. Variance support enables implicit conversion of classes that implement these interfaces.

Starting with .NET Framework 4, the following interfaces are variant:

- IEnumerable<T> (T is covariant)
- IEnumerator<T> (T is covariant)
- IQueryable<T> (T is covariant)
- IGrouping<TKey,TElement> (`TKey` and `TElement` are covariant)
- IComparer<T> (T is contravariant)
- IEqualityComparer<T> (T is contravariant)
- IComparable<T> (T is contravariant)

Starting with .NET Framework 4.5, the following interfaces are variant:

- IReadOnlyList<T> (T is covariant)
- IReadOnlyCollection<T> (T is covariant)

Covariance permits a method to have a more derived return type than that defined by the generic type parameter of the interface. To illustrate the covariance feature, consider these generic interfaces: `IEnumerable<Object>` and `IEnumerable<String>`. The `IEnumerable<String>` interface does not inherit the `IEnumerable<Object>` interface. However, the `String` type does inherit the `Object` type, and in some cases you may want to assign objects of these interfaces to each other. This is shown in the following code example.

C#

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

In earlier versions of .NET Framework, this code causes a compilation error in C# and, if `Option Strict` is on, in Visual Basic. But now you can use `strings` instead of `objects`, as shown in the previous example, because the IEnumerable<T> interface is covariant.

Contravariance permits a method to have argument types that are less derived than that specified by the generic parameter of the interface. To illustrate contravariance, assume that you have created a `BaseComparer` class to compare instances of the `BaseClass` class. The `BaseComparer` class implements the `IEqualityComparer<BaseClass>` interface. Because the IEqualityComparer<T> interface is now contravariant, you can use `BaseComparer` to compare instances of classes that inherit the `BaseClass` class. This is shown in the following code example.

C#

```
// Simple hierarchy of classes.
class BaseClass { }
class DerivedClass : BaseClass { }

// Comparer class.
class BaseComparer : IEqualityComparer<BaseClass>
{
    public int GetHashCode(BaseClass baseInstance)
    {
        return baseInstance.GetHashCode();
    }
    public bool Equals(BaseClass x, BaseClass y)
    {
        return x == y;
    }
}
class Program
{
    static void Test()
    {
        IEqualityComparer<BaseClass> baseComparer = new BaseComparer();

        // Implicit conversion of IEqualityComparer<BaseClass> to
        // IEqualityComparer<DerivedClass>.
        IEqualityComparer<DerivedClass> childComparer = baseComparer;
    }
}
```

For more examples, see Using Variance in Interfaces for Generic Collections (C#).

Variance in generic interfaces is supported for reference types only. Value types do not support variance. For example, `IEnumerable<int>` cannot be implicitly converted to `IEnumerable<object>`, because integers are represented by a value type.

C#

```
IEnumerable<int> integers = new List<int>();
// The following statement generates a compiler error,
```

```
// because int is a value type.
// IEnumerable<Object> objects = integers;
```

It is also important to remember that classes that implement variant interfaces are still invariant. For example, although List<T> implements the covariant interface IEnumerable<T>, you cannot implicitly convert `List<String>` to `List<Object>`. This is illustrated in the following code example.

C#

```
// The following line generates a compiler error
// because classes are invariant.
// List<Object> list = new List<String>();

// You can use the interface object instead.
IEnumerable<Object> listObjects = new List<String>();
```

See also

- Using Variance in Interfaces for Generic Collections (C#)
- Creating Variant Generic Interfaces (C#)
- Generic Interfaces
- Variance in Delegates (C#)

Creating Variant Generic Interfaces (C#)

Article • 09/15/2021

You can declare generic type parameters in interfaces as covariant or contravariant. *Covariance* allows interface methods to have more derived return types than that defined by the generic type parameters. *Contravariance* allows interface methods to have argument types that are less derived than that specified by the generic parameters. A generic interface that has covariant or contravariant generic type parameters is called *variant*.

> ⓘ **Note**
>
> .NET Framework 4 introduced variance support for several existing generic interfaces. For the list of the variant interfaces in .NET, see Variance in Generic Interfaces (C#).

Declaring Variant Generic Interfaces

You can declare variant generic interfaces by using the `in` and `out` keywords for generic type parameters.

> ⓘ **Important**
>
> `ref`, `in`, and `out` parameters in C# cannot be variant. Value types also do not support variance.

You can declare a generic type parameter covariant by using the `out` keyword. The covariant type must satisfy the following conditions:

- The type is used only as a return type of interface methods and not used as a type of method arguments. This is illustrated in the following example, in which the type `R` is declared covariant.

```
C#
```

```
interface ICovariant<out R>
{
    R GetSomething();
    // The following statement generates a compiler error.
    // void SetSomething(R sampleArg);
```

```
}
```

There is one exception to this rule. If you have a contravariant generic delegate as a method parameter, you can use the type as a generic type parameter for the delegate. This is illustrated by the type `R` in the following example. For more information, see Variance in Delegates (C#) and Using Variance for Func and Action Generic Delegates (C#).

```
C#

interface ICovariant<out R>
{
    void DoSomething(Action<R> callback);
}
```

- The type is not used as a generic constraint for the interface methods. This is illustrated in the following code.

```
C#

interface ICovariant<out R>
{
    // The following statement generates a compiler error
    // because you can use only contravariant or invariant types
    // in generic constraints.
    // void DoSomething<T>() where T : R;
}
```

You can declare a generic type parameter contravariant by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a return type of interface methods. The contravariant type can also be used for generic constraints. The following code shows how to declare a contravariant interface and use a generic constraint for one of its methods.

```
C#

interface IContravariant<in A>
{
    void SetSomething(A sampleArg);
    void DoSomething<T>() where T : A;
    // The following statement generates a compiler error.
    // A GetSomething();
}
```

It is also possible to support both covariance and contravariance in the same interface, but for different type parameters, as shown in the following code example.

C#

```
interface IVariant<out R, in A>
{
    R GetSomething();
    void SetSomething(A sampleArg);
    R GetSetSomethings(A sampleArg);
}
```

Implementing Variant Generic Interfaces

You implement variant generic interfaces in classes by using the same syntax that is used for invariant interfaces. The following code example shows how to implement a covariant interface in a generic class.

C#

```
interface ICovariant<out R>
{
    R GetSomething();
}
class SampleImplementation<R> : ICovariant<R>
{
    public R GetSomething()
    {
        // Some code.
        return default(R);
    }
}
```

Classes that implement variant interfaces are invariant. For example, consider the following code.

C#

```
// The interface is covariant.
ICovariant<Button> ibutton = new SampleImplementation<Button>();
ICovariant<Object> iobj = ibutton;

// The class is invariant.
SampleImplementation<Button> button = new SampleImplementation<Button>();
// The following statement generates a compiler error
// because classes are invariant.
// SampleImplementation<Object> obj = button;
```

Extending Variant Generic Interfaces

When you extend a variant generic interface, you have to use the `in` and `out` keywords to explicitly specify whether the derived interface supports variance. The compiler does not infer the variance from the interface that is being extended. For example, consider the following interfaces.

C#

```
interface ICovariant<out T> { }
interface IInvariant<T> : ICovariant<T> { }
interface IExtCovariant<out T> : ICovariant<T> { }
```

In the `IInvariant<T>` interface, the generic type parameter `T` is invariant, whereas in `IExtCovariant<out T>` the type parameter is covariant, although both interfaces extend the same interface. The same rule is applied to contravariant generic type parameters.

You can create an interface that extends both the interface where the generic type parameter `T` is covariant and the interface where it is contravariant if in the extending interface the generic type parameter `T` is invariant. This is illustrated in the following code example.

C#

```
interface ICovariant<out T> { }
interface IContravariant<in T> { }
interface IInvariant<T> : ICovariant<T>, IContravariant<T> { }
```

However, if a generic type parameter `T` is declared covariant in one interface, you cannot declare it contravariant in the extending interface, or vice versa. This is illustrated in the following code example.

C#

```
interface ICovariant<out T> { }
// The following statement generates a compiler error.
// interface ICoContraVariant<in T> : ICovariant<T> { }
```

Avoiding Ambiguity

When you implement variant generic interfaces, variance can sometimes lead to ambiguity. Such ambiguity should be avoided.

For example, if you explicitly implement the same variant generic interface with different generic type parameters in one class, it can create ambiguity. The compiler does not produce an error in this case, but it's not specified which interface implementation will be chosen at run time. This ambiguity could lead to subtle bugs in your code. Consider the following code example.

C#

```
// Simple class hierarchy.
class Animal { }
class Cat : Animal { }
class Dog : Animal { }

// This class introduces ambiguity
// because IEnumerable<out T> is covariant.
class Pets : IEnumerable<Cat>, IEnumerable<Dog>
{
    IEnumerator<Cat> IEnumerable<Cat>.GetEnumerator()
    {
        Console.WriteLine("Cat");
        // Some code.
        return null;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        // Some code.
        return null;
    }

    IEnumerator<Dog> IEnumerable<Dog>.GetEnumerator()
    {
        Console.WriteLine("Dog");
        // Some code.
        return null;
    }
}
class Program
{
    public static void Test()
    {
        IEnumerable<Animal> pets = new Pets();
        pets.GetEnumerator();
    }
}
```

In this example, it is unspecified how the `pets.GetEnumerator` method chooses between `Cat` and `Dog`. This could cause problems in your code.

See also

- Variance in Generic Interfaces (C#)
- Using Variance for Func and Action Generic Delegates (C#)

Using Variance in Interfaces for Generic Collections (C#)

Article • 09/15/2021

A covariant interface allows its methods to return more derived types than those specified in the interface. A contravariant interface allows its methods to accept parameters of less derived types than those specified in the interface.

In .NET Framework 4, several existing interfaces became covariant and contravariant. These include IEnumerable<T> and IComparable<T>. This enables you to reuse methods that operate with generic collections of base types for collections of derived types.

For a list of variant interfaces in .NET, see Variance in Generic Interfaces (C#).

Converting Generic Collections

The following example illustrates the benefits of covariance support in the IEnumerable<T> interface. The `PrintFullName` method accepts a collection of the `IEnumerable<Person>` type as a parameter. However, you can reuse it for a collection of the `IEnumerable<Employee>` type because `Employee` inherits `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

class Program
{
    // The method has a parameter of the IEnumerable<Person> type.
    public static void PrintFullName(IEnumerable<Person> persons)
    {
        foreach (Person person in persons)
        {
            Console.WriteLine("Name: {0} {1}",
            person.FirstName, person.LastName);
        }
    }
```

```
    public static void Test()
    {
        IEnumerable<Employee> employees = new List<Employee>();

        // You can pass IEnumerable<Employee>,
        // although the method expects IEnumerable<Person>.

        PrintFullName(employees);

    }
}
```

Comparing Generic Collections

The following example illustrates the benefits of contravariance support in the IEqualityComparer<T> interface. The `PersonComparer` class implements the `IEqualityComparer<Person>` interface. However, you can reuse this class to compare a sequence of objects of the `Employee` type because `Employee` inherits `Person`.

C#

```
// Simple hierarchy of classes.
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}

public class Employee : Person { }

// The custom comparer for the Person type
// with standard implementations of Equals()
// and GetHashCode() methods.
class PersonComparer : IEqualityComparer<Person>
{
    public bool Equals(Person x, Person y)
    {
        if (Object.ReferenceEquals(x, y)) return true;
        if (Object.ReferenceEquals(x, null) ||
            Object.ReferenceEquals(y, null))
            return false;
        return x.FirstName == y.FirstName && x.LastName == y.LastName;
    }
    public int GetHashCode(Person person)
    {
        if (Object.ReferenceEquals(person, null)) return 0;
        int hashFirstName = person.FirstName == null
            ? 0 : person.FirstName.GetHashCode();
        int hashLastName = person.LastName.GetHashCode();
        return hashFirstName ^ hashLastName;
```

```
    }
}

class Program
{

    public static void Test()
    {
        List<Employee> employees = new List<Employee> {
               new Employee() {FirstName = "Michael", LastName =
"Alexander"},
               new Employee() {FirstName = "Jeff", LastName = "Price"}
            };

        // You can pass PersonComparer,
        // which implements IEqualityComparer<Person>,
        // although the method expects IEqualityComparer<Employee>.

        IEnumerable<Employee> noduplicates =
            employees.Distinct<Employee>(new PersonComparer());

        foreach (var employee in noduplicates)
            Console.WriteLine(employee.FirstName + " " + employee.LastName);
    }
}
```

See also

- Variance in Generic Interfaces (C#)

Variance in Delegates (C#)

Article • 09/15/2021

.NET Framework 3.5 introduced variance support for matching method signatures with delegate types in all delegates in C#. This means that you can 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. This includes both generic and non-generic delegates.

For example, consider the following code, which has two classes and two delegates: generic and non-generic.

C#

```
public class First { }
public class Second : First { }
public delegate First SampleDelegate(Second a);
public delegate R SampleGenericDelegate<A, R>(A a);
```

When you create delegates of the `SampleDelegate` or `SampleGenericDelegate<A, R>` types, you can assign any one of the following methods to those delegates.

C#

```
// Matching signature.
public static First ASecondRFirst(Second second)
{ return new First(); }

// The return type is more derived.
public static Second ASecondRSecond(Second second)
{ return new Second(); }

// The argument type is less derived.
public static First AFirstRFirst(First first)
{ return new First(); }

// The return type is more derived
// and the argument type is less derived.
public static Second AFirstRSecond(First first)
{ return new Second(); }
```

The following code example illustrates the implicit conversion between the method signature and the delegate type.

C#

```
// Assigning a method with a matching signature
// to a non-generic delegate. No conversion is necessary.
SampleDelegate dNonGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a non-generic delegate.
// The implicit conversion is used.
SampleDelegate dNonGenericConversion = AFirstRSecond;

// Assigning a method with a matching signature to a generic delegate.
// No conversion is necessary.
SampleGenericDelegate<Second, First> dGeneric = ASecondRFirst;
// Assigning a method with a more derived return type
// and less derived argument type to a generic delegate.
// The implicit conversion is used.
SampleGenericDelegate<Second, First> dGenericConversion = AFirstRSecond;
```

For more examples, see Using Variance in Delegates (C#) and Using Variance for Func and Action Generic Delegates (C#).

Variance in Generic Type Parameters

In .NET Framework 4 or later you can enable implicit conversion between delegates, so that generic delegates that have different types specified by generic type parameters can be assigned to each other, if the types are inherited from each other as required by variance.

To enable implicit conversion, you must explicitly declare generic parameters in a delegate as covariant or contravariant by using the `in` or `out` keyword.

The following code example shows how you can create a delegate that has a covariant generic type parameter.

C#

```
// Type T is declared covariant by using the out keyword.
public delegate T SampleGenericDelegate <out T>();

public static void Test()
{
    SampleGenericDelegate <String> dString = () => " ";

    // You can assign delegates to each other,
    // because the type T is declared covariant.
    SampleGenericDelegate <Object> dObject = dString;
}
```

If you use only variance support to match method signatures with delegate types and do not use the `in` and `out` keywords, you may find that sometimes you can instantiate delegates with identical lambda expressions or methods, but you cannot assign one delegate to another.

In the following code example, `SampleGenericDelegate<String>` cannot be explicitly converted to `SampleGenericDelegate<Object>`, although `String` inherits `Object`. You can fix this problem by marking the generic parameter `T` with the `out` keyword.

C#

```
public delegate T SampleGenericDelegate<T>();

public static void Test()
{
    SampleGenericDelegate<String> dString = () => " ";

    // You can assign the dObject delegate
    // to the same lambda expression as dString delegate
    // because of the variance support for
    // matching method signatures with delegate types.
    SampleGenericDelegate<Object> dObject = () => " ";

    // The following statement generates a compiler error
    // because the generic type T is not marked as covariant.
    // SampleGenericDelegate <Object> dObject = dString;

}
```

Generic Delegates That Have Variant Type Parameters in .NET

.NET Framework 4 introduced variance support for generic type parameters in several existing generic delegates:

- `Action` delegates from the System namespace, for example, Action<T> and Action<T1,T2>
- `Func` delegates from the System namespace, for example, Func<TResult> and Func<T,TResult>
- The Predicate<T> delegate
- The Comparison<T> delegate
- The Converter<TInput,TOutput> delegate

For more information and examples, see Using Variance for Func and Action Generic Delegates (C#).

Declaring Variant Type Parameters in Generic Delegates

If a generic delegate has covariant or contravariant generic type parameters, it can be referred to as a *variant generic delegate*.

You can declare a generic type parameter covariant in a generic delegate by using the `out` keyword. The covariant type can be used only as a method return type and not as a type of method arguments. The following code example shows how to declare a covariant generic delegate.

```
C#

public delegate R DCovariant<out R>();
```

You can declare a generic type parameter contravariant in a generic delegate by using the `in` keyword. The contravariant type can be used only as a type of method arguments and not as a method return type. The following code example shows how to declare a contravariant generic delegate.

```
C#

public delegate void DContravariant<in A>(A a);
```

> **Important**
>
> `ref`, `in`, and `out` parameters in C# can't be marked as variant.

It is also possible to support both variance and covariance in the same delegate, but for different type parameters. This is shown in the following example.

```
C#

public delegate R DVariant<in A, out R>(A a);
```

Instantiating and Invoking Variant Generic Delegates

You can instantiate and invoke variant delegates just as you instantiate and invoke invariant delegates. In the following example, the delegate is instantiated by a lambda

expression.

```
C#

DVariant<String, String> dvariant = (String str) => str + " ";
dvariant("test");
```

Combining Variant Generic Delegates

Don't combine variant delegates. The Combine method does not support variant delegate conversion and expects delegates to be of exactly the same type. This can lead to a run-time exception when you combine delegates either by using the Combine method or by using the `+` operator, as shown in the following code example.

```
C#

Action<object> actObj = x => Console.WriteLine("object: {0}", x);
Action<string> actStr = x => Console.WriteLine("string: {0}", x);
// All of the following statements throw exceptions at run time.
// Action<string> actCombine = actStr + actObj;
// actStr += actObj;
// Delegate.Combine(actStr, actObj);
```

Variance in Generic Type Parameters for Value and Reference Types

Variance for generic type parameters is supported for reference types only. For example, `DVariant<int>` can't be implicitly converted to `DVariant<Object>` or `DVariant<long>`, because integer is a value type.

The following example demonstrates that variance in generic type parameters is not supported for value types.

```
C#

// The type T is covariant.
public delegate T DVariant<out T>();

// The type T is invariant.
public delegate T DInvariant<T>();

public static void Test()
{
    int i = 0;
```

```
    DInvariant<int> dInt = () => i;
    DVariant<int> dVariantInt = () => i;

    // All of the following statements generate a compiler error
    // because type variance in generic parameters is not supported
    // for value types, even if generic type parameters are declared
variant.
    // DInvariant<Object> dObject = dInt;
    // DInvariant<long> dLong = dInt;
    // DVariant<Object> dVariantObject = dVariantInt;
    // DVariant<long> dVariantLong = dVariantInt;
}
```

See also

- Generics
- Using Variance for Func and Action Generic Delegates (C#)
- How to combine delegates (Multicast Delegates)

Using Variance in Delegates (C#)

07/03/2025

When you assign a method to a delegate, *covariance* and *contravariance* provide flexibility for matching a delegate type with a method signature. Covariance permits a method to have return type that is more derived than that defined in the delegate. Contravariance permits a method that has parameter types that are less derived than those in the delegate type.

Example 1: Covariance

Description

This example demonstrates how delegates can be used with methods that have return types that are derived from the return type in the delegate signature. The data type returned by `DogsHandler` is of type `Dogs`, which derives from the `Mammals` type that is defined in the delegate.

Code

C#

```
class Mammals {}
class Dogs : Mammals {}

class Program
{
    // Define the delegate.
    public delegate Mammals HandlerMethod();

    public static Mammals MammalsHandler()
    {
        return null;
    }

    public static Dogs DogsHandler()
    {
        return null;
    }

    static void Test()
    {
        HandlerMethod handlerMammals = MammalsHandler;

        // Covariance enables this assignment.
        HandlerMethod handlerDogs = DogsHandler;
```

```
    }
}
```

Example 2: Contravariance

Description

This example demonstrates how delegates can be used with methods that have parameters whose types are base types of the delegate signature parameter type. With contravariance, you can use one event handler instead of separate handlers. The following example makes use of two delegates:

- A custom `KeyEventHandler` delegate that defines the signature of a key event. Its signature is:

  C#

  ```
  public delegate void KeyEventHandler(object sender, KeyEventArgs e)
  ```

- A custom `MouseEventHandler` delegate that defines the signature of a mouse event. Its signature is:

  C#

  ```
  public delegate void MouseEventHandler(object sender, MouseEventArgs e)
  ```

The example defines an event handler with an EventArgs parameter and uses it to handle both key and mouse events. This works because EventArgs is a base type of both the custom `KeyEventArgs` and `MouseEventArgs` classes defined in the example. Contravariance allows a method that accepts a base type parameter to be used for events that provide derived type parameters.

How contravariance works in this example

When you subscribe to an event, the compiler checks if your event handler method is compatible with the event's delegate signature. With contravariance:

1. The `KeyDown` event expects a method that takes `KeyEventArgs`.
2. The `MouseClick` event expects a method that takes `MouseEventArgs`.
3. Your `MultiHandler` method takes the base type `EventArgs`.

4. Since `KeyEventArgs` and `MouseEventArgs` both inherit from `EventArgs`, they can be safely passed to a method expecting `EventArgs`.
5. The compiler allows this assignment because it's safe - the `MultiHandler` can work with any `EventArgs` instance.

This is contravariance in action: you can use a method with a "less specific" (base type) parameter where a "more specific" (derived type) parameter is expected.

Code

C#

```
// Custom EventArgs classes to demonstrate the hierarchy
public class KeyEventArgs(string keyCode) : EventArgs
{
    public string KeyCode { get; set; } = keyCode;
}

public class MouseEventArgs(int x, int y) : EventArgs
{
    public int X { get; set; } = x;
    public int Y { get; set; } = y;
}

// Define delegate types that match the Windows Forms pattern
public delegate void KeyEventHandler(object sender, KeyEventArgs e);
public delegate void MouseEventHandler(object sender, MouseEventArgs e);

// A simple class that demonstrates contravariance with events
public class Button
{
    // Events that expect specific EventArgs-derived types
    public event KeyEventHandler? KeyDown;
    public event MouseEventHandler? MouseClick;

    // Method to simulate key press
    public void SimulateKeyPress(string key)
    {
        Console.WriteLine($"Simulating key press: {key}");
        KeyDown?.Invoke(this, new KeyEventArgs(key));
    }

    // Method to simulate mouse click
    public void SimulateMouseClick(int x, int y)
    {
        Console.WriteLine($"Simulating mouse click at ({x}, {y})");
        MouseClick?.Invoke(this, new MouseEventArgs(x, y));
    }
}

public class Form1
```

```
{
    private Button button1;

    public Form1()
    {
        button1 = new Button();

        // Event handler that accepts a parameter of the base EventArgs type.
        // This method can handle events that expect more specific EventArgs-
derived types
        // due to contravariance in delegate parameters.

        // You can use a method that has an EventArgs parameter,
        // although the KeyDown event expects the KeyEventArgs parameter.
        button1.KeyDown += MultiHandler;

        // You can use the same method for an event that expects
        // the MouseEventArgs parameter.
        button1.MouseClick += MultiHandler;
    }

    // Event handler that accepts a parameter of the base EventArgs type.
    // This works for both KeyDown and MouseClick events because:
    // - KeyDown expects KeyEventHandler(object sender, KeyEventArgs e)
    // - MouseClick expects MouseEventHandler(object sender, MouseEventArgs e)
    // - Both KeyEventArgs and MouseEventArgs derive from EventArgs
    // - Contravariance allows a method with a base type parameter (EventArgs)
    //   to be used where a derived type parameter is expected
    private void MultiHandler(object sender, EventArgs e)
    {
        Console.WriteLine($"MultiHandler called at: {DateTime.Now:HH:mm:ss.fff}");

        // You can check the actual type of the event args if needed
        switch (e)
        {
            case KeyEventArgs keyArgs:
                Console.WriteLine($"  - Key event: {keyArgs.KeyCode}");
                break;
            case MouseEventArgs mouseArgs:
                Console.WriteLine($"  - Mouse event: ({mouseArgs.X},
{mouseArgs.Y})");
                break;
            default:
                Console.WriteLine($"  - Generic event: {e.GetType().Name}");
                break;
        }
    }

    public void DemonstrateEvents()
    {
        Console.WriteLine("Demonstrating contravariance in event handlers:");
        Console.WriteLine("Same MultiHandler method handles both events!\n");

        button1.SimulateKeyPress("Enter");
        button1.SimulateMouseClick(100, 200);
```

```
        button1.SimulateKeyPress("Escape");
        button1.SimulateMouseClick(50, 75);
    }
}
```

Key points about contravariance

C#

```
// Demonstration of how contravariance works with delegates:
//
// 1. KeyDown event signature: KeyEventHandler(object sender, KeyEventArgs e)
//    where KeyEventArgs derives from EventArgs
//
// 2. MouseClick event signature: MouseEventHandler(object sender, MouseEventArgs
e)
//    where MouseEventArgs derives from EventArgs
//
// 3. Our MultiHandler method signature: MultiHandler(object sender, EventArgs e)
//
// 4. Contravariance allows us to use MultiHandler (which expects EventArgs)
//    for events that provide more specific types (KeyEventArgs, MouseEventArgs)
//    because the more specific types can be safely treated as their base type.
//
// This is safe because:
// - The MultiHandler only uses members available on the base EventArgs type
// - KeyEventArgs and MouseEventArgs can be implicitly converted to EventArgs
// - The compiler knows that any EventArgs-derived type can be passed safely
```

When you run this example, you'll see that the same `MultiHandler` method successfully handles both key and mouse events, demonstrating how contravariance enables more flexible and reusable event handling code.

See also

- Variance in Delegates (C#)
- Using Variance for Func and Action Generic Delegates (C#)

Using Variance for Func and Action Generic Delegates (C#)

08/06/2025

These examples demonstrate how to use covariance and contravariance in the `Func` and `Action` generic delegates to enable reuse of methods and provide more flexibility in your code.

For more information about covariance and contravariance, see Variance in Delegates (C#).

Using Delegates with Covariant Type Parameters

The following example illustrates the benefits of covariance support in the generic `Func` delegates. The `FindByTitle` method takes a parameter of the `String` type and returns an object of the `Employee` type. However, you can assign this method to the `Func<String, Person>` delegate because `Employee` inherits `Person`.

C#

```
// Simple hierarchy of classes.
public class Person { }
public class Employee : Person { }
class Program
{
    static Employee FindByTitle(String title)
    {
        // This is a stub for a method that returns
        // an employee that has the specified title.
        return new Employee();
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Func<String, Employee> findEmployee = FindByTitle;

        // The delegate expects a method to return Person,
        // but you can assign it a method that returns Employee.
        Func<String, Person> findPerson = FindByTitle;

        // You can also assign a delegate
        // that returns a more derived type
        // to a delegate that returns a less derived type.
        findPerson = findEmployee;

    }
}
```

Using Delegates with Contravariant Type Parameters

The following example illustrates the benefits of contravariance support in the generic `Action` delegates. The `AddToContacts` method takes a parameter of the `Person` type. However, you can assign this method to the `Action<Employee>` delegate because `Employee` inherits `Person`.

C#

```
public class Person { }
public class Employee : Person { }
class Program
{
    static void AddToContacts(Person person)
    {
        // This method adds a Person object
        // to a contact list.
    }

    static void Test()
    {
        // Create an instance of the delegate without using variance.
        Action<Person> addPersonToContacts = AddToContacts;

        // The Action delegate expects
        // a method that has an Employee parameter,
        // but you can assign it a method that has a Person parameter
        // because Employee derives from Person.
        Action<Employee> addEmployeeToContacts = AddToContacts;

        // You can also assign a delegate
        // that accepts a less derived parameter to a delegate
        // that accepts a more derived parameter.
        addEmployeeToContacts = addPersonToContacts;
    }
}
```

Contravariance and anonymous functions

When working with anonymous functions (lambda expressions), you might encounter counterintuitive behavior related to contravariance. Consider the following example:

C#

```
public class Person
{
    public virtual void ReadContact() { /*...*/ }
}
```

```
public class Employee : Person
{
    public override void ReadContact() { /*...*/ }
}

class Program
{
    private static void Main()
    {
        var personReadContact = (Person p) => p.ReadContact();

        // This works - contravariance allows assignment.
        Action<Employee> employeeReadContact = personReadContact;

        // This causes a compile error: CS1661.
        // Action<Employee> employeeReadContact2 = (Person p) => p.ReadContact();
    }
}
```

This behavior seems contradictory: if contravariance allows assigning a delegate that accepts a base type (`Person`) to a delegate variable expecting a derived type (`Employee`), why does direct assignment of the lambda expression fail?

The key difference is **type inference**. In the first case, the lambda expression is first assigned to a variable with type `var`, which causes the compiler to infer the lambda's type as `Action<Person>`. The subsequent assignment to `Action<Employee>` succeeds because of contravariance rules for delegates.

In the second case, the compiler cannot directly infer that the lambda expression `(Person p) => p.ReadContact()` should have type `Action<Person>` when it's being assigned to `Action<Employee>`. The type inference rules for anonymous functions don't automatically apply contravariance during the initial type determination.

Workarounds

To make direct assignment work, you can use explicit casting:

C#

```
// Explicit cast to the desired delegate type.
Action<Employee> employeeReadContact = (Action<Person>)((Person p) =>
p.ReadContact());

// Or specify the lambda parameter type that matches the target delegate.
Action<Employee> employeeReadContact2 = (Employee e) => e.ReadContact();
```

This behavior illustrates the difference between delegate contravariance (which works after types are established) and lambda expression type inference (which occurs during compilation).

See also

- Covariance and Contravariance (C#)
- Generics

Iterators (C#)

Article • 04/12/2023

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.

C#

```
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.

C#

```
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.

C#

```
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.

C#

```
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 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.

C#

```
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)
    {
        Console.Write("{0} ", number);
    }
    Console.WriteLine();
    // Output: 0 1 2 3 4 5 6 7 8 9

    foreach (int number in theStack.TopN(7))
    {
        Console.Write("{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--)
        {
            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 `IEnumerable<T>` 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 common 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 in .NET.

- 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
- Generics

Statements (C# Programming Guide)

Article • 04/22/2023

The actions that a program takes are expressed in statements. Common actions include declaring variables, assigning values, calling methods, looping through collections, and branching to one or another block of code, depending on a given condition. The order in which statements are executed in a program is called the flow of control or flow of execution. The flow of control may vary every time that a program is run, depending on how the program reacts to input that it receives at run time.

A statement can consist of a single line of code that ends in a semicolon, or a series of single-line statements in a block. A statement block is enclosed in {} brackets and can contain nested blocks. The following code shows two examples of single-line statements, and a multi-line statement block:

C#

```
    public static void Main()
    {
        // Declaration statement.
        int counter;

        // Assignment statement.
        counter = 1;

        // Error! This is an expression, not an expression statement.
        // counter + 1;

        // Declaration statements with initializers are functionally
        // equivalent to  declaration statement followed by assignment
statement:
        int[] radii = [15, 32, 108, 74, 9]; // Declare and initialize an
array.
        const double pi = 3.14159; // Declare and initialize  constant.

        // foreach statement block that contains multiple statements.
        foreach (int radius in radii)
        {
            // Declaration statement with initializer.
            double circumference = pi * (2 * radius);

            // Expression statement (method invocation). A single-line
            // statement can span multiple text lines because line breaks
            // are treated as white space, which is ignored by the compiler.
            System.Console.WriteLine($"Radius of circle #{counter} is
{radius}. Circumference = {circumference:N2}");

            // Expression statement (postfix increment).
            counter++;
```

```
        } // End of foreach statement block
    } // End of Main method body.
} // End of SimpleStatements class.
/*
   Output:
    Radius of circle #1 = 15. Circumference = 94.25
    Radius of circle #2 = 32. Circumference = 201.06
    Radius of circle #3 = 108. Circumference = 678.58
    Radius of circle #4 = 74. Circumference = 464.96
    Radius of circle #5 = 9. Circumference = 56.55
*/
```

Types of statements

The following table lists the various types of statements in C# and their associated keywords, with links to topics that include more information:

Expand table

Category	C# keywords / notes
Declaration statements	A declaration statement introduces a new variable or constant. A variable declaration can optionally assign a value to the variable. In a constant declaration, the assignment is required.
Expression statements	Expression statements that calculate a value must store the value in a variable.
Selection statements	Selection statements enable you to branch to different sections of code, depending on one or more specified conditions. For more information, see the following topics: • if • switch
Iteration statements	Iteration statements enable you to loop through collections like arrays, or perform the same set of statements repeatedly until a specified condition is met. For more information, see the following topics: • do • for • foreach • while
Jump statements	Jump statements transfer control to another section of code. For more information, see the following topics: • break • continue • goto

Category	C# keywords / notes
	• return • yield
Exception-handling statements	Exception-handling statements enable you to gracefully recover from exceptional conditions that occur at run time. For more information, see the following topics: • throw • try-catch • try-finally • try-catch-finally
checked and unchecked	The `checked` and `unchecked` statements enable you to specify whether integral-type numerical operations are allowed to cause an overflow when the result is stored in a variable that is too small to hold the resulting value.
The `await` statement	If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an `await` expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method. For a simple example, see the "Async Methods" section of Methods. For more information, see Asynchronous Programming with async and await.
The `yield return` statement	An iterator performs a custom iteration over a collection, such as a list or an array. An iterator 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 when the iterator is called the next time. For more information, see Iterators.
The `fixed` statement	The fixed statement prevents the garbage collector from relocating a movable variable. For more information, see fixed.
The `lock` statement	The lock statement enables you to limit access to blocks of code to only one thread at a time. For more information, see lock.
Labeled statements	You can give a statement a label and then use the goto keyword to jump to the labeled statement. (See the example in the following row.)
The empty statement	The empty statement consists of a single semicolon. It does nothing and can be used in places where a statement is required but no action needs to be performed.

Declaration statements

The following code shows examples of variable declarations with and without an initial assignment, and a constant declaration with the necessary initialization.

```
C#

// Variable declaration statements.
double area;
double radius = 2;

// Constant declaration statement.
const double pi = 3.14159;
```

Expression statements

The following code shows examples of expression statements, including assignment, object creation with assignment, and method invocation.

```
C#

// Expression statement (assignment).
area = 3.14 * (radius * radius);

// Error. Not  statement because no assignment:
//circ * 2;

// Expression statement (method invocation).
System.Console.WriteLine();

// Expression statement (new object creation).
System.Collections.Generic.List<string> strings =
    new System.Collections.Generic.List<string>();
```

The empty statement

The following examples show two uses for an empty statement:

```
C#

void ProcessMessages()
{
    while (ProcessMessage())
        ; // Statement needed here.
}

void F()
{
    //...
```

```
    if (done) goto exit;
//...
exit:
    ; // Statement needed here.
}
```

Embedded statements

Some statements, for example, iteration statements, always have an embedded statement that follows them. This embedded statement may be either a single statement or multiple statements enclosed by {} brackets in a statement block. Even single-line embedded statements can be enclosed in {} brackets, as shown in the following example:

C#

```
// Recommended style. Embedded statement in  block.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
{
    System.Console.WriteLine(s);
}

// Not recommended.
foreach (string s in System.IO.Directory.GetDirectories(
                        System.Environment.CurrentDirectory))
    System.Console.WriteLine(s);
```

An embedded statement that is not enclosed in {} brackets cannot be a declaration statement or a labeled statement. This is shown in the following example:

C#

```
if(pointB == true)
    //Error CS1023:
    int radius = 5;
```

Put the embedded statement in a block to fix the error:

C#

```
if (b == true)
{
    // OK:
    System.DateTime d = System.DateTime.Now;
```

```
    System.Console.WriteLine(d.ToLongDateString());
}
```

Nested statement blocks

Statement blocks can be nested, as shown in the following code:

C#

```
foreach (string s in System.IO.Directory.GetDirectories(
    System.Environment.CurrentDirectory))
{
    if (s.StartsWith("CSharp"))
    {
        if (s.EndsWith("TempFolder"))
        {
            return s;
        }
    }
}
return "Not found.";
```

Unreachable statements

If the compiler determines that the flow of control can never reach a particular statement under any circumstances, it will produce warning CS0162, as shown in the following example:

C#

```
// An over-simplified example of unreachable code.
const int val = 5;
if (val < 4)
{
    System.Console.WriteLine("I'll never write anything."); //CS0162
}
```

C# language specification

For more information, see the Statements section of the C# language specification.

See also

- Statement keywords
- C# operators and expressions

Expression-bodied members (C# programming guide)

07/03/2025

Expression body definitions let you provide a member's implementation in a concise, readable form. You can use an expression body definition whenever the logic for any supported member, such as a method or property, consists of a single expression. An expression body definition has the following general syntax:

```
C#

member => expression;
```

where *expression* is a valid expression.

Expression body definitions can be used with the following type members:

- Method
- Read-only property
- Property
- Constructor
- Finalizer
- Indexer

Methods

An expression-bodied method consists of a single expression that returns a value whose type matches the method's return type, or, for methods that return `void`, that performs some operation. For example, types that override the ToString method typically include a single expression that returns the string representation of the current object.

The following example defines a `Person` class that overrides the ToString method with an expression body definition. It also defines a `DisplayName` method that displays a name to the console. Additionally, it includes several methods that take parameters, demonstrating how expression-bodied members work with method parameters. The `return` keyword is not used in any of the expression body definitions.

```
C#

using System;
```

```
namespace ExpressionBodiedMembers;

public class Person
{
   public Person(string firstName, string lastName)
   {
      fname = firstName;
      lname = lastName;
   }

   private string fname;
   private string lname;

   public override string ToString() => $"{fname} {lname}".Trim();
   public void DisplayName() => Console.WriteLine(ToString());

   // Expression-bodied methods with parameters
   public string GetFullName(string title) => $"{title} {fname} {lname}";
   public int CalculateAge(int birthYear) => DateTime.Now.Year - birthYear;
   public bool IsOlderThan(int age) => CalculateAge(1990) > age;
   public string FormatName(string format) => format.Replace("{first}",
fname).Replace("{last}", lname);
}

class Example
{
   public static void Main()
   {
      Person p = new Person("Mandy", "Dejesus");
      Console.WriteLine(p);
      p.DisplayName();

      // Examples with parameters
      Console.WriteLine(p.GetFullName("Dr."));
      Console.WriteLine($"Age: {p.CalculateAge(1990)}");
      Console.WriteLine($"Is older than 25: {p.IsOlderThan(25)}");
      Console.WriteLine(p.FormatName("Last: {last}, First: {first}"));
   }
}
```

For more information, see Methods (C# Programming Guide).

Read-only properties

You can use expression body definition to implement a read-only property. To do that, use the following syntax:

C#

```
PropertyType PropertyName => expression;
```

The following example defines a `Location` class whose read-only `Name` property is implemented as an expression body definition that returns the value of the private `locationName` field:

C#

```
public class Location
{
   private string locationName;

   public Location(string name)
   {
      locationName = name;
   }

   public string Name => locationName;
}
```

For more information about properties, see Properties (C# Programming Guide).

Properties

You can use expression body definitions to implement property `get` and `set` accessors. The following example demonstrates how to do that:

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}

// Example with multiple parameters
public class Point
{
   public double X { get; }
   public double Y { get; }

   // Constructor with multiple parameters
   public Point(double x, double y) => (X, Y) = (x, y);
```

```
    // Constructor with single parameter (creates point at origin on axis)
    public Point(double coordinate) => (X, Y) = (coordinate, 0);
}
```

For more information about properties, see Properties (C# Programming Guide).

Events

Similarly, event `add` and `remove` accessors can be expression-bodied:

C#

```
public class ChangedEventArgs : EventArgs
{
    public required int NewValue { get; init; }
}

public class ObservableNum(int _value)
{
    public event EventHandler<ChangedEventArgs> ChangedGeneric = default!;

    public event EventHandler Changed
    {
        // Note that, while this is syntactically valid, it won't work as expected
because it's creating a new delegate object with each call.
        add => ChangedGeneric += (sender, args) => value(sender, args);
        remove => ChangedGeneric -= (sender, args) => value(sender, args);
    }

    public int Value
    {
        get => _value;
        set => ChangedGeneric?.Invoke(this, new() { NewValue = (_value = value) });
    }
}
```

For more information about events, see Events (C# Programming Guide).

Constructors

An expression body definition for a constructor typically consists of a single assignment expression or a method call that handles the constructor's arguments or initializes instance state.

The following example defines a `Location` class whose constructor has a single string parameter named *name*. The expression body definition assigns the argument to the `Name`

property. The example also shows a `Point` class with constructors that take multiple parameters, demonstrating how expression-bodied constructors work with different parameter combinations.

C#

```
public class Location
{
   private string locationName;

   public Location(string name) => Name = name;

   public string Name
   {
      get => locationName;
      set => locationName = value;
   }
}

// Example with multiple parameters
public class Point
{
   public double X { get; }
   public double Y { get; }

   // Constructor with multiple parameters
   public Point(double x, double y) => (X, Y) = (x, y);

   // Constructor with single parameter (creates point at origin on axis)
   public Point(double coordinate) => (X, Y) = (coordinate, 0);
}
```

For more information, see Constructors (C# Programming Guide).

Finalizers

An expression body definition for a finalizer typically contains cleanup statements, such as statements that release unmanaged resources.

The following example defines a finalizer that uses an expression body definition to indicate that the finalizer has been called.

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;
```

```
    ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is executing.");
}
```

For more information, see Finalizers (C# Programming Guide).

Indexers

Like with properties, indexer `get` and `set` accessors consist of expression body definitions if the `get` accessor consists of a single expression that returns a value or the `set` accessor performs a simple assignment.

The following example defines a class named `Sports` that includes an internal String array that contains the names of some sports. Both the indexer `get` and `set` accessors are implemented as expression body definitions.

C#

```
using System;
using System.Collections.Generic;

namespace SportsExample;

public class Sports
{
   private string[] types = [ "Baseball", "Basketball", "Football",
                              "Hockey", "Soccer", "Tennis",
                              "Volleyball" ];

   public string this[int i]
   {
      get => types[i];
      set => types[i] = value;
   }
}
```

For more information, see Indexers (C# Programming Guide).

See also

- .NET code style rules for expression-bodied-members

Equality comparisons (C# Programming Guide)

Article • 03/12/2024

It is sometimes necessary to compare two values for equality. In some cases, you are testing for *value equality*, also known as *equivalence*, which means that the values that are contained by the two variables are equal. In other cases, you have to determine whether two variables refer to the same underlying object in memory. This type of equality is called *reference equality*, or *identity*. This topic describes these two kinds of equality and provides links to other topics for more information.

Reference equality

Reference equality means that two object references refer to the same underlying object. This can occur through simple assignment, as shown in the following example.

C#

```
using System;
class Test
{
    public int Num { get; set; }
    public string Str { get; set; }

    public static void Main()
    {
        Test a = new Test() { Num = 1, Str = "Hi" };
        Test b = new Test() { Num = 1, Str = "Hi" };

        bool areEqual = System.Object.ReferenceEquals(a, b);
        // False:
        System.Console.WriteLine($"ReferenceEquals(a, b) = {areEqual}");

        // Assign b to a.
        b = a;

        // Repeat calls with different results.
        areEqual = System.Object.ReferenceEquals(a, b);
        // True:
        System.Console.WriteLine($"ReferenceEquals(a, b) = {areEqual}");
    }
}
```

In this code, two objects are created, but after the assignment statement, both references refer to the same object. Therefore they have reference equality. Use the

ReferenceEquals method to determine whether two references refer to the same object.

The concept of reference equality applies only to reference types. Value type objects cannot have reference equality because when an instance of a value type is assigned to a variable, a copy of the value is made. Therefore you can never have two unboxed structs that refer to the same location in memory. Furthermore, if you use ReferenceEquals to compare two value types, the result will always be `false`, even if the values that are contained in the objects are all identical. This is because each variable is boxed into a separate object instance. For more information, see How to test for reference equality (Identity).

Value equality

Value equality means that two objects contain the same value or values. For primitive value types such as int or bool, tests for value equality are straightforward. You can use the == operator, as shown in the following example.

```
C#
```

```
int a = GetOriginalValue();
int b = GetCurrentValue();

// Test for value equality.
if (b == a)
{
    // The two integers are equal.
}
```

For most other types, testing for value equality is more complex because it requires that you understand how the type defines it. For classes and structs that have multiple fields or properties, value equality is often defined to mean that all fields or properties have the same value. For example, two `Point` objects might be defined to be equivalent if pointA.X is equal to pointB.X and pointA.Y is equal to pointB.Y. For records, value equality means that two variables of a record type are equal if the types match and all property and field values match.

However, there is no requirement that equivalence be based on all the fields in a type. It can be based on a subset. When you compare types that you do not own, you should make sure to understand specifically how equivalence is defined for that type. For more information about how to define value equality in your own classes and structs, see How to define value equality for a type.

Value equality for floating-point values

Equality comparisons of floating-point values (double and float) are problematic because of the imprecision of floating-point arithmetic on binary computers. For more information, see the remarks in the topic System.Double.

Related topics

Expand table

Title	Description
How to test for reference equality (Identity)	Describes how to determine whether two variables have reference equality.
How to define value equality for a type	Describes how to provide a custom definition of value equality for a type.
Types	Provides information about the C# type system and links to additional information.
Records	Provides information about record types, which test for value equality by default.

How to define value equality for a class or struct (C# Programming Guide)

08/30/2025

> **Tip**
>
> **Consider using records first.** Records automatically implement value equality with minimal code, making them the recommended approach for most data-focused types. If you need custom value equality logic or cannot use records, continue with the manual implementation steps below.

When you define a class or struct, you decide whether it makes sense to create a custom definition of value equality (or equivalence) for the type. Typically, you implement value equality when you expect to add objects of the type to a collection, or when their primary purpose is to store a set of fields or properties. You can base your definition of value equality on a comparison of all the fields and properties in the type, or you can base the definition on a subset.

In either case, and in both classes and structs, your implementation should follow the five guarantees of equivalence (for the following rules, assume that `x`, `y` and `z` are not null):

1. The reflexive property: `x.Equals(x)` returns `true`.
2. The symmetric property: `x.Equals(y)` returns the same value as `y.Equals(x)`.
3. The transitive property: if `(x.Equals(y) && y.Equals(z))` returns `true`, then `x.Equals(z)` returns `true`.
4. Successive invocations of `x.Equals(y)` return the same value as long as the objects referenced by x and y aren't modified.
5. Any non-null value isn't equal to null. However, `x.Equals(y)` throws an exception when `x` is null. That breaks rules 1 or 2, depending on the argument to `Equals`.

Any struct that you define already has a default implementation of value equality that it inherits from the System.ValueType override of the Object.Equals(Object) method. This implementation uses reflection to examine all the fields and properties in the type. Although this implementation produces correct results, it is relatively slow compared to a custom implementation that you write specifically for the type.

The implementation details for value equality are different for classes and structs. However, both classes and structs require the same basic steps for implementing equality:

1. **Override the virtual Object.Equals(Object) method.** This provides polymorphic equality behavior, allowing your objects to be compared correctly when treated as `object` references. It ensures proper behavior in collections and when using polymorphism. In most cases, your implementation of `bool Equals( object obj )` should just call into the type-specific `Equals` method that is the implementation of the System.IEquatable<T> interface. (See step 2.)

2. **Implement the System.IEquatable<T> interface by providing a type-specific `Equals` method.** This provides type-safe equality checking without boxing, resulting in better performance. It also avoids unnecessary casting and enables compile-time type checking. This is where the actual equivalence comparison is performed. For example, you might decide to define equality by comparing only one or two fields in your type. Don't throw exceptions from `Equals`. For classes that are related by inheritance:

   - This method should examine only fields that are declared in the class. It should call `base.Equals` to examine fields that are in the base class. (Don't call `base.Equals` if the type inherits directly from Object, because the Object implementation of Object.Equals(Object) performs a reference equality check.)

   - Two variables should be deemed equal only if the run-time types of the variables being compared are the same. Also, make sure that the `IEquatable` implementation of the `Equals` method for the run-time type is used if the run-time and compile-time types of a variable are different. One strategy for making sure run-time types are always compared correctly is to implement `IEquatable` only in `sealed` classes. For more information, see the class example later in this article.

3. **Optional but recommended: Overload the == and != operators.** This provides consistent and intuitive syntax for equality comparisons, matching user expectations from built-in types. It ensures that `obj1 == obj2` and `obj1.Equals(obj2)` behave the same way.

4. **Override Object.GetHashCode so that two objects that have value equality produce the same hash code.** This is required for correct behavior in hash-based collections like `Dictionary<TKey,TValue>` and `HashSet<T>`. Objects that are equal must have equal hash codes, or these collections won't work correctly.

5. **Optional: To support definitions for "greater than" or "less than," implement the IComparable<T> interface for your type, and also overload the <= and >= operators.** This enables sorting operations and provides a complete ordering relationship for your type, useful when adding objects to sorted collections or when sorting arrays or lists.

Record example

The following example shows how records automatically implement value equality with minimal code. The first record `TwoDPoint` is a simple record type that automatically implements value equality. The second record `ThreeDPoint` demonstrates that records can be derived from other records and still maintain proper value equality behavior:

C#

```
namespace ValueEqualityRecord;

public record TwoDPoint(int X, int Y);

public record ThreeDPoint(int X, int Y, int Z) : TwoDPoint(X, Y);

class Program
{
    static void Main(string[] args)
    {
        // Create some points
        TwoDPoint pointA = new TwoDPoint(3, 4);
        TwoDPoint pointB = new TwoDPoint(3, 4);
        TwoDPoint pointC = new TwoDPoint(5, 6);

        ThreeDPoint point3D_A = new ThreeDPoint(3, 4, 5);
        ThreeDPoint point3D_B = new ThreeDPoint(3, 4, 5);
        ThreeDPoint point3D_C = new ThreeDPoint(3, 4, 7);

        Console.WriteLine("=== Value Equality with Records ===");

        // Value equality works automatically
        Console.WriteLine($"pointA.Equals(pointB) = {pointA.Equals(pointB)}"); //
True
        Console.WriteLine($"pointA == pointB = {pointA == pointB}"); // True
        Console.WriteLine($"pointA.Equals(pointC) = {pointA.Equals(pointC)}"); //
False
        Console.WriteLine($"pointA == pointC = {pointA == pointC}"); // False

        Console.WriteLine("\n=== Hash Codes ===");

        // Equal objects have equal hash codes automatically
        Console.WriteLine($"pointA.GetHashCode() = {pointA.GetHashCode()}");
        Console.WriteLine($"pointB.GetHashCode() = {pointB.GetHashCode()}");
        Console.WriteLine($"pointC.GetHashCode() = {pointC.GetHashCode()}");

        Console.WriteLine("\n=== Inheritance with Records ===");

        // Inheritance works correctly with value equality
        Console.WriteLine($"point3D_A.Equals(point3D_B) =
{point3D_A.Equals(point3D_B)}"); // True
        Console.WriteLine($"point3D_A == point3D_B = {point3D_A == point3D_B}");
// True
```

```
        Console.WriteLine($"point3D_A.Equals(point3D_C) =
{point3D_A.Equals(point3D_C)}"); // False

        // Different types are not equal (unlike problematic class example)
        Console.WriteLine($"pointA.Equals(point3D_A) =
{pointA.Equals(point3D_A)}"); // False

        Console.WriteLine("\n=== Collections ===");

        // Works seamlessly with collections
        var pointSet = new HashSet<TwoDPoint> { pointA, pointB, pointC };
        Console.WriteLine($"Set contains {pointSet.Count} unique points"); // 2
unique points

        var pointDict = new Dictionary<TwoDPoint, string>
        {
            { pointA, "First point" },
            { pointC, "Different point" }
        };

        // Demonstrate that equivalent points work as the same key
        var duplicatePoint = new TwoDPoint(3, 4);
        Console.WriteLine($"Dictionary contains key for {duplicatePoint}:
{pointDict.ContainsKey(duplicatePoint)}"); // True
        Console.WriteLine($"Dictionary contains {pointDict.Count} entries"); // 2
entries

        Console.WriteLine("\n=== String Representation ===");

        // Automatic ToString implementation
        Console.WriteLine($"pointA.ToString() = {pointA}");
        Console.WriteLine($"point3D_A.ToString() = {point3D_A}");

        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Expected Output:
=== Value Equality with Records ===
pointA.Equals(pointB) = True
pointA == pointB = True
pointA.Equals(pointC) = False
pointA == pointC = False

=== Hash Codes ===
pointA.GetHashCode() = -1400834708
pointB.GetHashCode() = -1400834708
pointC.GetHashCode() = -148136000

=== Inheritance with Records ===
point3D_A.Equals(point3D_B) = True
point3D_A == point3D_B = True
point3D_A.Equals(point3D_C) = False
pointA.Equals(point3D_A) = False
```

```
=== Collections ===
Set contains 2 unique points
Dictionary contains key for TwoDPoint { X = 3, Y = 4 }: True
Dictionary contains 2 entries

=== String Representation ===
pointA.ToString() = TwoDPoint { X = 3, Y = 4 }
point3D_A.ToString() = ThreeDPoint { X = 3, Y = 4, Z = 5 }
*/
```

Records provide several advantages for value equality:

- **Automatic implementation**: Records automatically implement System.IEquatable<T> and override Object.Equals, Object.GetHashCode, and the `==`/`!=` operators.
- **Correct inheritance behavior**: Records implement `IEquatable<T>` using virtual methods that check the runtime type of both operands, ensuring correct behavior in inheritance hierarchies and polymorphic scenarios.
- **Immutability by default**: Records encourage immutable design, which works well with value equality semantics.
- **Concise syntax**: Positional parameters provide a compact way to define data types.
- **Better performance**: The compiler-generated equality implementation is optimized and doesn't use reflection like the default struct implementation.

Use records when your primary goal is to store data and you need value equality semantics.

Records with members that use reference equality

When records contain members that use reference equality, the automatic value equality behavior of records doesn't work as expected. This applies to collections like System.Collections.Generic.List<T>, arrays, and other reference types that don't implement value-based equality (with the notable exception of System.String, which does implement value equality).

> **ⓘ Important**
>
> While records provide excellent value equality for basic data types, they don't automatically solve value equality for members that use reference equality. If a record contains a System.Collections.Generic.List<T>, System.Array, or other reference types that don't implement value equality, two record instances with identical content in those members will still not be equal because the members use reference equality.

C#

```
public record PersonWithHobbies(string Name, List<string> Hobbies);

var person1 = new PersonWithHobbies("Alice", new List<string> { "Reading",
"Swimming" });
var person2 = new PersonWithHobbies("Alice", new List<string> { "Reading",
"Swimming" });

Console.WriteLine(person1.Equals(person2)); // False - different List
instances!
```

This is because records use the Object.Equals method of each member, and collection types typically use reference equality rather than comparing their contents.

The following shows the problem:

C#

```
// Records with reference-equality members don't work as expected
public record PersonWithHobbies(string Name, List<string> Hobbies);
```

Here's how this behaves when you run the code:

C#

```
Console.WriteLine("=== Records with Collections - The Problem ===");

// Problem: Records with mutable collections use reference equality for the
collection
var person1 = new PersonWithHobbies("Alice", [ "Reading", "Swimming" ]);
var person2 = new PersonWithHobbies("Alice", [ "Reading", "Swimming" ]);

Console.WriteLine($"person1: {person1}");
Console.WriteLine($"person2: {person2}");
Console.WriteLine($"person1.Equals(person2): {person1.Equals(person2)}"); //
False! Different List instances
Console.WriteLine($"Lists have same content:
{person1.Hobbies.SequenceEqual(person2.Hobbies)}"); // True
Console.WriteLine();
```

Solutions for records with reference-equality members

- **Custom System.IEquatable<T> implementation**: Replace the compiler-generated equality with a hand-coded version that provides content-based comparison for reference-equality members. For collections, implement element-by-element comparison using Enumerable.SequenceEqual or similar methods.

- **Use value types where possible**: Consider if your data can be represented with value types or immutable structures that naturally support value equality, such as System.Numerics.Vector<T> or Plane.
- **Use types with value-based equality**: For collections, consider using types that implement value-based equality or implement custom collection types that override Object.Equals to provide content-based comparison, such as System.Collections.Immutable.ImmutableArray<T> or System.Collections.Immutable.ImmutableList<T>.
- **Design with reference equality in mind**: Accept that some members will use reference equality and design your application logic accordingly, ensuring that you reuse the same instances when equality is important.

Here's an example of implementing custom equality for records with collections:

C#

```
// A potential solution using IEquatable<T> with custom equality
public record PersonWithHobbiesFixed(string Name, List<string> Hobbies) :
IEquatable<PersonWithHobbiesFixed>
{
    public virtual bool Equals(PersonWithHobbiesFixed? other)
    {
        if (ReferenceEquals(null, other)) return false;
        if (ReferenceEquals(this, other)) return true;

        // Use SequenceEqual for List comparison
        return Name == other.Name && Hobbies.SequenceEqual(other.Hobbies);
    }

    public override int GetHashCode()
    {
        // Create hash based on content, not reference
        var hashCode = new HashCode();
        hashCode.Add(Name);
        foreach (var hobby in Hobbies)
        {
            hashCode.Add(hobby);
        }
        return hashCode.ToHashCode();
    }
}
```

This custom implementation works correctly:

C#

```
Console.WriteLine("=== Solution 1: Custom IEquatable Implementation ===");

var personFixed1 = new PersonWithHobbiesFixed("Bob", [ "Cooking", "Hiking" ]);
var personFixed2 = new PersonWithHobbiesFixed("Bob", [ "Cooking", "Hiking" ]);

Console.WriteLine($"personFixed1: {personFixed1}");
Console.WriteLine($"personFixed2: {personFixed2}");
Console.WriteLine($"personFixed1.Equals(personFixed2):
{personFixed1.Equals(personFixed2)}"); // True! Custom equality
Console.WriteLine();
```

The same issue affects arrays and other collection types:

C#

```
// These also use reference equality - the issue persists
public record PersonWithHobbiesArray(string Name, string[] Hobbies);

public record PersonWithHobbiesImmutable(string Name, IReadOnlyList<string>
Hobbies);
```

Arrays also use reference equality, producing the same unexpected results:

C#

```
Console.WriteLine("=== Arrays Also Use Reference Equality ===");

var personArray1 = new PersonWithHobbiesArray("Charlie", ["Gaming", "Music" ]);
var personArray2 = new PersonWithHobbiesArray("Charlie", ["Gaming", "Music" ]);

Console.WriteLine($"personArray1: {personArray1}");
Console.WriteLine($"personArray2: {personArray2}");
Console.WriteLine($"personArray1.Equals(personArray2):
{personArray1.Equals(personArray2)}"); // False! Arrays use reference equality too
Console.WriteLine($"Arrays have same content:
{personArray1.Hobbies.SequenceEqual(personArray2.Hobbies)}"); // True
Console.WriteLine();
```

Even readonly collections exhibit this reference equality behavior:

C#

```
Console.WriteLine("=== Same Issue with IReadOnlyList ===");

var personImmutable1 = new PersonWithHobbiesImmutable("Diana", [ "Art", "Travel"
]);
var personImmutable2 = new PersonWithHobbiesImmutable("Diana", [ "Art", "Travel"
]);
```

```
Console.WriteLine($"personImmutable1: {personImmutable1}");
Console.WriteLine($"personImmutable2: {personImmutable2}");
Console.WriteLine($"personImmutable1.Equals(personImmutable2):
{personImmutable1.Equals(personImmutable2)}"); // False! Reference equality
Console.WriteLine($"Content is the same:
{personImmutable1.Hobbies.SequenceEqual(personImmutable2.Hobbies)}"); // True
Console.WriteLine();
```

The key insight is that records solve the *structural* equality problem but don't change the *semantic* equality behavior of the types they contain.

Class example

The following example shows how to implement value equality in a class (reference type). This manual approach is needed when you can't use records or need custom equality logic:

C#

```
namespace ValueEqualityClass;

class TwoDPoint : IEquatable<TwoDPoint>
{
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
    {
        if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.X = x;
        this.Y = y;
    }

    public override bool Equals(object obj) => this.Equals(obj as TwoDPoint);

    public bool Equals(TwoDPoint p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // If run-time types are not exactly the same, return false.
```

```
        if (this.GetType() != p.GetType())
        {
            return false;
        }

        // Return true if the fields match.
        // Note that the base class is not invoked because it is
        // System.Object, which defines Equals as reference equality.
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs == rhs);
}

// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object obj) => this.Equals(obj as ThreeDPoint);

    public bool Equals(ThreeDPoint p)
    {
        if (p is null)
        {
            return false;
        }
```

```
        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Check properties that this class declares.
        if (Z == p.Z)
        {
            // Let base class check its own fields
            // and do the run-time type comparison.
            return base.Equals((TwoDPoint)p);
        }
        else
        {
            return false;
        }
    }

    public override int GetHashCode() => (X, Y, Z).GetHashCode();

    public static bool operator ==(ThreeDPoint lhs, ThreeDPoint rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(ThreeDPoint lhs, ThreeDPoint rhs) => !(lhs ==
rhs);
}

class Program
{
    static void Main(string[] args)
    {
        ThreeDPoint pointA = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointB = new ThreeDPoint(3, 4, 5);
        ThreeDPoint pointC = null;
        int i = 5;

        Console.WriteLine($"pointA.Equals(pointB) = {pointA.Equals(pointB)}");
        Console.WriteLine($"pointA == pointB = {pointA == pointB}");
        Console.WriteLine($"null comparison = {pointA.Equals(pointC)}");
```

```
        Console.WriteLine($"Compare to some other type = {pointA.Equals(i)}");

        TwoDPoint pointD = null;
        TwoDPoint pointE = null;

        Console.WriteLine($"Two null TwoDPoints are equal: {pointD == pointE}");

        pointE = new TwoDPoint(3, 4);
        Console.WriteLine($"(pointE == pointA) = {pointE == pointA}");
        Console.WriteLine($"(pointA == pointE) = {pointA == pointE}");
        Console.WriteLine($"(pointA != pointE) = {pointA != pointE}");

        System.Collections.ArrayList list = new System.Collections.ArrayList();
        list.Add(new ThreeDPoint(3, 4, 5));
        Console.WriteLine($"pointE.Equals(list[0]): {pointE.Equals(list[0])}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}

/* Output:
    pointA.Equals(pointB) = True
    pointA == pointB = True
    null comparison = False
    Compare to some other type = False
    Two null TwoDPoints are equal: True
    (pointE == pointA) = False
    (pointA == pointE) = False
    (pointA != pointE) = True
    pointE.Equals(list[0]): False
*/
```

On classes (reference types), the default implementation of both Object.Equals(Object) methods performs a reference equality comparison, not a value equality check. When an implementer overrides the virtual method, the purpose is to give it value equality semantics.

The `==` and `!=` operators can be used with classes even if the class does not overload them. However, the default behavior is to perform a reference equality check. In a class, if you overload the `Equals` method, you should overload the `==` and `!=` operators, but it is not required.

> **ⓘ Important**
>
> The preceding example code may not handle every inheritance scenario the way you expect. Consider the following code:
>
> ```
> C#
> ```

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
Console.WriteLine(p1.Equals(p2)); // output: True
```

This code reports that `p1` equals `p2` despite the difference in `z` values. The difference is ignored because the compiler picks the `TwoDPoint` implementation of `IEquatable` based on the compile-time type. This is a fundamental issue with polymorphic equality in inheritance hierarchies.

Polymorphic equality

When implementing value equality in inheritance hierarchies with classes, the standard approach shown in the class example can lead to incorrect behavior when objects are used polymorphically. The issue occurs because System.IEquatable<T> implementations are chosen based on compile-time type, not runtime type.

The problem with standard implementations

Consider this problematic scenario:

C#

```
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);  // Declared as TwoDPoint
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);  // Declared as TwoDPoint
Console.WriteLine(p1.Equals(p2)); // True - but should be False!
```

The comparison returns `True` because the compiler selects `TwoDPoint.Equals(TwoDPoint)` based on the declared type, ignoring the `Z` coordinate differences.

The key to correct polymorphic equality is ensuring that all equality comparisons use the virtual Object.Equals method, which can check runtime types and handle inheritance correctly. This can be achieved by using explicit interface implementation for System.IEquatable<T> that delegates to the virtual method:

The base class demonstrates the key patterns:

C#

```
// Safe polymorphic equality implementation using explicit interface
implementation
class TwoDPoint : IEquatable<TwoDPoint>
{
```

```
    public int X { get; private set; }
    public int Y { get; private set; }

    public TwoDPoint(int x, int y)
    {
        if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.X = x;
        this.Y = y;
    }

    public override bool Equals(object? obj) => Equals(obj as TwoDPoint);

    // Explicit interface implementation prevents compile-time type issues
    bool IEquatable<TwoDPoint>.Equals(TwoDPoint? p) => Equals((object?)p);

    protected virtual bool Equals(TwoDPoint? p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // If run-time types are not exactly the same, return false.
        if (this.GetType() != p.GetType())
        {
            return false;
        }

        // Return true if the fields match.
        // Note that the base class is not invoked because it is
        // System.Object, which defines Equals as reference equality.
        return (X == p.X) && (Y == p.Y);
    }

    public override int GetHashCode() => (X, Y).GetHashCode();

    public static bool operator ==(TwoDPoint? lhs, TwoDPoint? rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                return true;
            }

            // Only the left side is null.
```

```
            return false;
        }
        // Equals handles case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(TwoDPoint? lhs, TwoDPoint? rhs) => !(lhs ==
rhs);
}
```

The derived class correctly extends the equality logic:

C#

```
// For the sake of simplicity, assume a ThreeDPoint IS a TwoDPoint.
class ThreeDPoint : TwoDPoint, IEquatable<ThreeDPoint>
{
    public int Z { get; private set; }

    public ThreeDPoint(int x, int y, int z)
        : base(x, y)
    {
        if ((z < 1) || (z > 2000))
        {
            throw new ArgumentException("Point must be in range 1 - 2000");
        }
        this.Z = z;
    }

    public override bool Equals(object? obj) => Equals(obj as ThreeDPoint);

    // Explicit interface implementation prevents compile-time type issues
    bool IEquatable<ThreeDPoint>.Equals(ThreeDPoint? p) => Equals((object?)p);

    protected override bool Equals(TwoDPoint? p)
    {
        if (p is null)
        {
            return false;
        }

        // Optimization for a common success case.
        if (Object.ReferenceEquals(this, p))
        {
            return true;
        }

        // Runtime type check happens in the base method
        if (p is ThreeDPoint threeD)
        {
            // Check properties that this class declares.
            if (Z != threeD.Z)
            {
```

```
                return false;
            }

            return base.Equals(p);
        }

        return false;
    }

    public override int GetHashCode() => (X, Y, Z).GetHashCode();

    public static bool operator ==(ThreeDPoint? lhs, ThreeDPoint? rhs)
    {
        if (lhs is null)
        {
            if (rhs is null)
            {
                // null == null = true.
                return true;
            }

            // Only the left side is null.
            return false;
        }
        // Equals handles the case of null on right side.
        return lhs.Equals(rhs);
    }

    public static bool operator !=(ThreeDPoint? lhs, ThreeDPoint? rhs) => !(lhs ==
rhs);
}
```

Here's how this implementation handles the problematic polymorphic scenarios:

C#

```
Console.WriteLine("=== Safe Polymorphic Equality ===");

// Test polymorphic scenarios that were problematic before
TwoDPoint p1 = new ThreeDPoint(1, 2, 3);
TwoDPoint p2 = new ThreeDPoint(1, 2, 4);
TwoDPoint p3 = new ThreeDPoint(1, 2, 3);
TwoDPoint p4 = new TwoDPoint(1, 2);

Console.WriteLine("Testing polymorphic equality (declared as TwoDPoint):");
Console.WriteLine($"p1 = ThreeDPoint(1, 2, 3) as TwoDPoint");
Console.WriteLine($"p2 = ThreeDPoint(1, 2, 4) as TwoDPoint");
Console.WriteLine($"p3 = ThreeDPoint(1, 2, 3) as TwoDPoint");
Console.WriteLine($"p4 = TwoDPoint(1, 2)");
Console.WriteLine();

Console.WriteLine($"p1.Equals(p2) = {p1.Equals(p2)}"); // False - different Z
values
```

```
Console.WriteLine($"p1.Equals(p3) = {p1.Equals(p3)}"); // True - same values
Console.WriteLine($"p1.Equals(p4) = {p1.Equals(p4)}"); // False - different types
Console.WriteLine($"p4.Equals(p1) = {p4.Equals(p1)}"); // False - different types
Console.WriteLine();
```

The implementation also correctly handles direct type comparisons:

C#

```
// Test direct type comparisons
var point3D_A = new ThreeDPoint(3, 4, 5);
var point3D_B = new ThreeDPoint(3, 4, 5);
var point3D_C = new ThreeDPoint(3, 4, 7);
var point2D_A = new TwoDPoint(3, 4);

Console.WriteLine("Testing direct type comparisons:");
Console.WriteLine($"point3D_A.Equals(point3D_B) = {point3D_A.Equals(point3D_B)}");
// True
Console.WriteLine($"point3D_A.Equals(point3D_C) = {point3D_A.Equals(point3D_C)}");
// False
Console.WriteLine($"point3D_A.Equals(point2D_A) = {point3D_A.Equals(point2D_A)}");
// False
Console.WriteLine($"point2D_A.Equals(point3D_A) = {point2D_A.Equals(point3D_A)}");
// False
Console.WriteLine();
```

The equality implementation also works properly with collections:

C#

```
// Test with collections
Console.WriteLine("Testing with collections:");
var hashSet = new HashSet<TwoDPoint> { p1, p2, p3, p4 };
Console.WriteLine($"HashSet contains {hashSet.Count} unique points"); // Should be
3: one ThreeDPoint(1,2,3), one ThreeDPoint(1,2,4), one TwoDPoint(1,2)

var dictionary = new Dictionary<TwoDPoint, string>
{
    { p1, "First 3D point" },
    { p2, "Second 3D point" },
    { p4, "2D point" }
};

Console.WriteLine($"Dictionary contains {dictionary.Count} entries");
Console.WriteLine($"Dictionary lookup for equivalent point:
{dictionary.ContainsKey(new ThreeDPoint(1, 2, 3))}"); // True
```

The preceding code demonstrates key elements to implementing value based equality:

- **Virtual `Equals(object?)` override**: The main equality logic happens in the virtual Object.Equals method, which is called regardless of compile-time type.
- **Runtime type checking**: Using `this.GetType() != p.GetType()` ensures that objects of different types are never considered equal.
- **Explicit interface implementation**: The System.IEquatable<T> implementation delegates to the virtual method, preventing compile-time type selection issues.
- **Protected virtual helper method**: The `protected virtual Equals(TwoDPoint? p)` method allows derived classes to override equality logic while maintaining type safety.

Use this pattern when:

- You have inheritance hierarchies where value equality is important
- Objects might be used polymorphically (declared as base type, instantiated as derived type)
- You need reference types with value equality semantics

The preferred approach is to use `record` types to implement value based equality. This approach requires a more complex implementation than the standard approach and requires thorough testing of polymorphic scenarios to ensure correctness.

Struct example

The following example shows how to implement value equality in a struct (value type). While structs have default value equality, a custom implementation can improve performance:

C#

```
namespace ValueEqualityStruct
{
    struct TwoDPoint : IEquatable<TwoDPoint>
    {
        public int X { get; private set; }
        public int Y { get; private set; }

        public TwoDPoint(int x, int y)
            : this()
        {
            if (x is (< 1 or > 2000) || y is (< 1 or > 2000))
            {
                throw new ArgumentException("Point must be in range 1 - 2000");
            }
            X = x;
            Y = y;
        }

        public override bool Equals(object? obj) => obj is TwoDPoint other &&
this.Equals(other);
```

```
        public bool Equals(TwoDPoint p) => X == p.X && Y == p.Y;

        public override int GetHashCode() => (X, Y).GetHashCode();

        public static bool operator ==(TwoDPoint lhs, TwoDPoint rhs) =>
lhs.Equals(rhs);

        public static bool operator !=(TwoDPoint lhs, TwoDPoint rhs) => !(lhs ==
rhs);
    }

    class Program
    {
        static void Main(string[] args)
        {
            TwoDPoint pointA = new TwoDPoint(3, 4);
            TwoDPoint pointB = new TwoDPoint(3, 4);
            int i = 5;

            // True:
            Console.WriteLine($"pointA.Equals(pointB) = {pointA.Equals(pointB)}");
            // True:
            Console.WriteLine($"pointA == pointB = {pointA == pointB}");
            // True:
            Console.WriteLine($"object.Equals(pointA, pointB) =
{object.Equals(pointA, pointB)}");
            // False:
            Console.WriteLine($"pointA.Equals(null) = {pointA.Equals(null)}");
            // False:
            Console.WriteLine($"(pointA == null) = {pointA == null}");
            // True:
            Console.WriteLine($"(pointA != null) = {pointA != null}");
            // False:
            Console.WriteLine($"pointA.Equals(i) = {pointA.Equals(i)}");
            // CS0019:
            // Console.WriteLine($"pointA == i = {pointA == i}");

            // Compare unboxed to boxed.
            System.Collections.ArrayList list = new
System.Collections.ArrayList();
            list.Add(new TwoDPoint(3, 4));
            // True:
            Console.WriteLine($"pointA.Equals(list[0]):
{pointA.Equals(list[0])}");

            // Compare nullable to nullable and to non-nullable.
            TwoDPoint? pointC = null;
            TwoDPoint? pointD = null;
            // False:
            Console.WriteLine($"pointA == (pointC = null) = {pointA == pointC}");
            // True:
            Console.WriteLine($"pointC == pointD = {pointC == pointD}");

            TwoDPoint temp = new TwoDPoint(3, 4);
```

```
            pointC = temp;
            // True:
            Console.WriteLine($"pointA == (pointC = 3,4) = {pointA == pointC}");

            pointD = temp;
            // True:
            Console.WriteLine($"pointD == (pointC = 3,4) = {pointD == pointC}");

            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }

    /* Output:
        pointA.Equals(pointB) = True
        pointA == pointB = True
        Object.Equals(pointA, pointB) = True
        pointA.Equals(null) = False
        (pointA == null) = False
        (pointA != null) = True
        pointA.Equals(i) = False
        pointE.Equals(list[0]): True
        pointA == (pointC = null) = False
        pointC == pointD = True
        pointA == (pointC = 3,4) = True
        pointD == (pointC = 3,4) = True
    */
}
```

For structs, the default implementation of Object.Equals(Object) (which is the overridden version in System.ValueType) performs a value equality check by using reflection to compare the values of every field in the type. Although this implementation produces correct results, it is relatively slow compared to a custom implementation that you write specifically for the type.

When you override the virtual `Equals` method in a struct, the purpose is to provide a more efficient means of performing the value equality check and optionally to base the comparison on some subset of the struct's fields or properties.

The == and != operators can't operate on a struct unless the struct explicitly overloads them.

See also

- Equality comparisons

ⓘ **Note:** The author created this article with assistance from AI. Learn more

How to test for reference equality (Identity) (C# Programming Guide)

Article • 01/31/2025

You do not have to implement any custom logic to support reference equality comparisons in your types. This functionality is provided for all types by the static Object.ReferenceEquals method.

The following example shows how to determine whether two variables have *reference equality*, which means that they refer to the same object in memory.

The example also shows why Object.ReferenceEquals always returns `false` for value types. This is due to **boxing**, which creates separate object instances for each value type argument. Additionally, you should not use ReferenceEquals to determine string equality.

Example

C#

```
using System.Text;

namespace TestReferenceEquality
{
    struct TestStruct
    {
        public int Num { get; private set; }
        public string Name { get; private set; }

        public TestStruct(int i, string s) : this()
        {
            Num = i;
            Name = s;
        }
    }

    class TestClass
    {
        public int Num { get; set; }
        public string? Name { get; set; }
    }

    class Program
    {
        static void Main()
        {
```

```
            // Demonstrate reference equality with reference types.
            #region ReferenceTypes

            // Create two reference type instances that have identical
values.
            TestClass tcA = new TestClass() { Num = 1, Name = "New
TestClass" };
            TestClass tcB = new TestClass() { Num = 1, Name = "New
TestClass" };

            Console.WriteLine($"ReferenceEquals(tcA, tcB) =
{Object.ReferenceEquals(tcA, tcB)}"); // false

            // After assignment, tcB and tcA refer to the same object.
            // They now have reference equality.
            tcB = tcA;
            Console.WriteLine($"After assignment: ReferenceEquals(tcA, tcB)
= {Object.ReferenceEquals(tcA, tcB)}"); // true

            // Changes made to tcA are reflected in tcB. Therefore, objects
            // that have reference equality also have value equality.
            tcA.Num = 42;
            tcA.Name = "TestClass 42";
            Console.WriteLine($"tcB.Name = {tcB.Name} tcB.Num: {tcB.Num}");
            #endregion

            // Demonstrate that two value type instances never have
reference equality.
            #region ValueTypes

            TestStruct tsC = new TestStruct( 1, "TestStruct 1");

            // Value types are boxed into separate objects when passed to
ReferenceEquals.
            // Even if the same variable is used twice, boxing ensures they
are different instances.
            TestStruct tsD = tsC;
            Console.WriteLine($"After assignment: ReferenceEquals(tsC, tsD)
= {Object.ReferenceEquals(tsC, tsD)}"); // false
            #endregion

            #region stringRefEquality
            // Constant strings within the same assembly are always interned
by the runtime.
            // This means they are stored in the same location in memory.
Therefore,
            // the two strings have reference equality although no
assignment takes place.
            string strA = "Hello world!";
            string strB = "Hello world!";
            Console.WriteLine($"ReferenceEquals(strA, strB) =
{Object.ReferenceEquals(strA, strB)}"); // true

            // After a new string is assigned to strA, strA and strB
            // are no longer interned and no longer have reference equality.
```

```
            strA = "Goodbye world!";
            Console.WriteLine($"strA = '{strA}' strB = '{strB}'");

            Console.WriteLine("After strA changes, ReferenceEquals(strA,
strB) = {0}",
                              Object.ReferenceEquals(strA, strB)); // false

            // A string that is created at runtime cannot be interned.
            StringBuilder sb = new StringBuilder("Hello world!");
            string stringC = sb.ToString();
            // False:
            Console.WriteLine($"ReferenceEquals(stringC, strB) =
{Object.ReferenceEquals(stringC, strB)}");

            // The string class overloads the == operator to perform an
equality comparison.
            Console.WriteLine($"stringC == strB = {stringC == strB}"); //
true

            #endregion

            // Keep the console open in debug mode.
            Console.WriteLine("Press any key to exit.");
            Console.ReadKey();
        }
    }
}

/* Output:
    ReferenceEquals(tcA, tcB) = False
    After assignment: ReferenceEquals(tcA, tcB) = True
    tcB.Name = TestClass 42 tcB.Num: 42
    After assignment: ReferenceEquals(tsC, tsD) = False
    ReferenceEquals(strA, strB) = True
    strA = "Goodbye world!" strB = "Hello world!"
    After strA changes, ReferenceEquals(strA, strB) = False
    ReferenceEquals(stringC, strB) = False
    stringC == strB = True
*/
```

The implementation of `Equals` in the System.Object universal base class also performs a reference equality check, but it is best not to use this because, if a class happens to override the method, the results might not be what you expect. The same is true for the `==` and `!=` operators. When they are operating on reference types, the default behavior of `==` and `!=` is to perform a reference equality check. However, derived classes can overload the operator to perform a value equality check. To minimize the potential for error, it is best to always use ReferenceEquals when you have to determine whether two objects have reference equality.

Constant strings within the same assembly are always interned by the runtime. That is, only one instance of each unique literal string is maintained. However, the runtime does not guarantee that strings created at run time are interned, nor does it guarantee that two equal constant strings in different assemblies are interned.

> **Note**
>
> `ReferenceEquals` returns `false` for value types due to **boxing**, as each argument is independently boxed into a separate object.

See also

- Equality Comparisons

Casting and type conversions (C# Programming Guide)

Article • 02/05/2025

Because C# is statically typed at compile time, after a variable is declared, it can't be declared again or assigned a value of another type unless that type is implicitly convertible to the variable's type. For example, the `string` can't be implicitly converted to `int`. Therefore, after you declare `i` as an `int`, you can't assign the string "Hello" to it, as the following code shows:

```
C#
```

```
int i;

// error CS0029: can't implicitly convert type 'string' to 'int'
i = "Hello";
```

However, you might sometimes need to copy a value into a variable or method parameter of another type. For example, you might have an integer variable that you need to pass to a method whose parameter is typed as `double`. Or you might need to assign a class variable to a variable of an interface type. These kinds of operations are called *type conversions*. In C#, you can perform the following kinds of conversions:

- **Implicit conversions**: No special syntax is required because the conversion always succeeds and no data is lost. Examples include conversions from smaller to larger integral types, conversions from derived classes to base classes, and span conversions.
- **Explicit conversions (casts)**: Explicit conversions require a cast expression. Casting is required when information might be lost in the conversion, or when the conversion might not succeed for other reasons. Typical examples include numeric conversion to a type that has less precision or a smaller range, and conversion of a base-class instance to a derived class.
- **User-defined conversions**: User-defined conversions use special methods that you can define to enable explicit and implicit conversions between custom types that don't have a base class–derived class relationship. For more information, see User-defined conversion operators.
- **Conversions with helper classes**: To convert between noncompatible types, such as integers and System.DateTime objects, or hexadecimal strings and byte arrays,

you can use the System.BitConverter class, the System.Convert class, and the `Parse` methods of the built-in numeric types, such as Int32.Parse. For more information, see the following articles:

- How to convert a byte array to an int
- How to convert a string to a number
- How to convert between hexadecimal strings and numeric types

Implicit conversions

For built-in numeric types, an implicit conversion can be made when the value to be stored can fit into the variable without being truncated or rounded off. For integral types, this restriction means the range of the source type is a proper subset of the range for the target type. For example, a variable of type long (64-bit integer) can store any value that an int (32-bit integer) can store. In the following example, the compiler implicitly converts the value of `num` on the right to a type `long` before assigning it to `bigNum`.

C#

```
// Implicit conversion. A long can
// hold any value an int can hold, and more!
int num = 2147483647;
long bigNum = num;
```

For a complete list of all implicit numeric conversions, see the Implicit numeric conversions section of the Built-in numeric conversions article.

For reference types, an implicit conversion always exists from a class to any one of its direct or indirect base classes or interfaces. No special syntax is necessary because a derived class always contains all the members of a base class.

C#

```
Derived d = new Derived();

// Always OK.
Base b = d;
```

Explicit conversions

However, if a conversion can't be made without a risk of losing information, the compiler requires that you perform an explicit conversion, which is called a *cast*. A cast is a way of explicitly making the conversion. It indicates you're aware data loss might occur, or the cast might fail at run time. To perform a cast, specify the destination type in parentheses before the expression you want converted. The following program casts a double to an int. The program doesn't compile without the cast.

C#

```
double x = 1234.7;
int a;
// Cast double to int.
a = (int)x;
Console.WriteLine(a);
// Output: 1234
```

For a complete list of supported explicit numeric conversions, see the Explicit numeric conversions section of the Built-in numeric conversions article.

For reference types, an explicit cast is required if you need to convert from a base type to a derived type:

A cast operation between reference types doesn't change the run-time type of the underlying object; it only changes the type of the value that is being used as a reference to that object. For more information, see Polymorphism.

Type conversion exceptions at run time

In some reference type conversions, the compiler can't determine whether a cast is valid. It's possible for a cast operation that compiles correctly to fail at run time. As shown in the following example, a type cast that fails at run time causes an InvalidCastException to be thrown.

C#

```
Animal a = new Mammal();
Reptile r = (Reptile)a; // InvalidCastException at run time
```

Explicitly casting the argument `a` to a `Reptile` makes a dangerous assumption. It's safer to not make assumptions, but rather check the type. C# provides the is operator to enable you to test for compatibility before actually performing a cast. For more information, see How to safely cast using pattern matching and the as and is operators.

C# language specification

For more information, see the Conversions section of the C# language specification.

See also

- Types
- Cast expression
- User-defined conversion operators
- Generalized Type Conversion
- How to convert a string to a number

Boxing and Unboxing (C# Programming Guide)

10/13/2025

Boxing is the process of converting a value type to the type `object` or to any interface type implemented by this value type. When the common language runtime (CLR) boxes a value type, it wraps the value inside a System.Object instance and stores it on the managed heap. Unboxing extracts the value type from the object. Boxing is implicit; unboxing is explicit. The concept of boxing and unboxing underlies the C# unified view of the type system in which a value of any type can be treated as an object.

In the following example, the integer variable `i` is *boxed* and assigned to object `o`.

C#

```
int i = 123;
// The following line boxes i.
object o = i;
```

The object `o` can then be unboxed and assigned to integer variable `i`:

C#

```
o = 123;
i = (int)o;  // unboxing
```

The following examples illustrate how boxing is used in C#.

C#

```
// String.Concat example.
// String.Concat has many versions. Rest the mouse pointer on
// Concat in the following statement to verify that the version
// that is used here takes three object arguments. Both 42 and
// true must be boxed.
Console.WriteLine(String.Concat("Answer", 42, true));

// List example.
// Create a list of objects to hold a heterogeneous collection
// of elements.
List<object> mixedList = [];

// Add a string element to the list.
mixedList.Add("First Group:");
```

```
// Add some integers to the list.
for (int j = 1; j < 5; j++)
{
    // Rest the mouse pointer over j to verify that you are adding
    // an int to a list of objects. Each element j is boxed when
    // you add j to mixedList.
    mixedList.Add(j);
}

// Add another string and more integers.
mixedList.Add("Second Group:");
for (int j = 5; j < 10; j++)
{
    mixedList.Add(j);
}

// Display the elements in the list. Declare the loop variable by
// using var, so that the compiler assigns its type.
foreach (var item in mixedList)
{
    // Rest the mouse pointer over item to verify that the elements
    // of mixedList are objects.
    Console.WriteLine(item);
}

// The following loop sums the squares of the first group of boxed
// integers in mixedList. The list elements are objects, and cannot
// be multiplied or added to the sum until they are unboxed. The
// unboxing must be done explicitly.
var sum = 0;
for (var j = 1; j < 5; j++)
{
    // The following statement causes a compiler error: Operator
    // '*' cannot be applied to operands of type 'object' and
    // 'object'.
    //sum += mixedList[j] * mixedList[j];

    // After the list elements are unboxed, the computation does
    // not cause a compiler error.
    sum += (int)mixedList[j] * (int)mixedList[j];
}

Console.WriteLine($"Sum: {sum}");

// Output:
// Answer42True
// First Group:
// 1
// 2
// 3
// 4
// Second Group:
// 5
// 6
// 7
```

```
// 8
// 9
// Sum: 30
```

Performance

In relation to simple assignments, boxing and unboxing are computationally expensive processes. When a value type is boxed, a new object must be allocated and constructed. To a lesser degree, the cast required for unboxing is also expensive computationally. For more information, see Performance.

Boxing

Boxing is used to store value types in the garbage-collected heap. Boxing is an implicit conversion of a value type to the type `object` or to any interface type implemented by this value type. Boxing a value type allocates an object instance on the heap and copies the value into the new object.

Consider the following declaration of a value-type variable:

C#

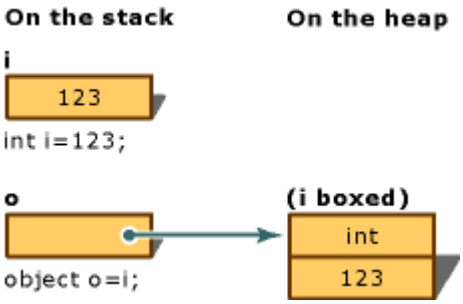
```
int i = 123;
```

The following statement implicitly applies the boxing operation on the variable `i`:

C#

```
// Boxing copies the value of i into object o.
object o = i;
```

The result of this statement is creating an object reference `o`, on the stack, that references a value of the type `int`, on the heap. This value is a copy of the value-type value assigned to the variable `i`. The difference between the two variables, `i` and `o`, is illustrated in the following image of boxing conversion:



It is also possible to perform the boxing explicitly as in the following example, but explicit boxing is never required:

C#

```
int i = 123;
object o = (object)i;  // explicit boxing
```

Example

This example converts an integer variable `i` to an object `o` by using boxing. Then, the value stored in the variable `i` is changed from `123` to `456`. The example shows that the original value type and the boxed object use separate memory locations, and therefore can store different values.

C#

```
        // Create an int variable
        int i = 123;

        // Box the value type into an object reference
        object o = i;  // boxing

        // Display the initial values
        Console.WriteLine($"Value of i: {i}");
        Console.WriteLine($"Value of boxed object o: {o}");

        // Modify the original value type
        i = 456;

        // Display the values after modification
        Console.WriteLine("\nAfter changing i to 456:");
        Console.WriteLine($"Value of i: {i}");
        Console.WriteLine($"Value of boxed object o: {o}");

        // Output:
        // Value of i: 123
        // Value of boxed object o: 123

        // After changing i to 456:
```

```
        // Value of i: 456
        // Value of boxed object o: 123
```

Unboxing

Unboxing is an explicit conversion from the type `object` to a value type or from an interface type to a value type that implements the interface. An unboxing operation consists of:

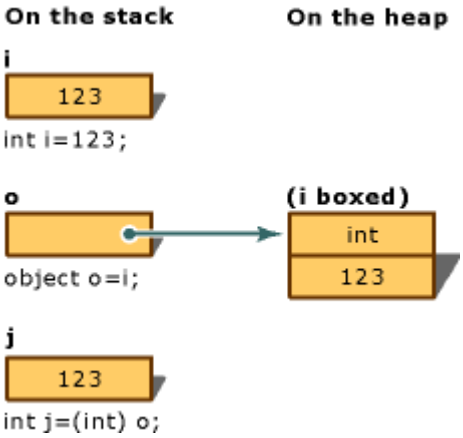
- Checking the object instance to make sure that it's a boxed value of the given value type.
- Copying the value from the instance into the value-type variable.

The following statements demonstrate both boxing and unboxing operations:

C#

```
int i = 123;      // a value type
object o = i;     // boxing
int j = (int)o;   // unboxing
```

The following figure demonstrates the result of the previous statements:



For the unboxing of value types to succeed at run time, the item being unboxed must be a reference to an object that was previously created by boxing an instance of that value type. Attempting to unbox `null` causes a NullReferenceException. Attempting to unbox a reference to an incompatible value type causes an InvalidCastException.

Example

The following example demonstrates a case of invalid unboxing and the resulting `InvalidCastException`. Using `try` and `catch`, an error message is displayed when the error occurs.

```
```

C#

```
class TestUnboxing
{
    static void Main()
    {
        int i = 123;
        object o = i;  // implicit boxing

        try
        {
            int j = (short)o;  // attempt to unbox

            Console.WriteLine("Unboxing OK.");
        }
        catch (InvalidCastException e)
        {
            Console.WriteLine($"{e.Message} Error: Incorrect unboxing.");
        }
    }
}
```

This program outputs:

```
Specified cast is not valid. Error: Incorrect unboxing.
```

If you change the statement:

C#

```
int j = (short)o;
```

To:

C#

```
int j = (int)o;
```

The conversion is performed, and you'll get the output:

```
Unboxing OK.
```

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Reference types
- Value types

How to convert a byte array to an int (C# Programming Guide)

Article • 09/23/2021

This example shows you how to use the BitConverter class to convert an array of bytes to an int and back to an array of bytes. You may have to convert from bytes to a built-in data type after you read bytes off the network, for example. In addition to the ToInt32(Byte[], Int32) method in the example, the following table lists methods in the BitConverter class that convert bytes (from an array of bytes) to other built-in types.

Expand table

Type returned	Method
`bool`	ToBoolean(Byte[], Int32)
`char`	ToChar(Byte[], Int32)
`double`	ToDouble(Byte[], Int32)
`short`	ToInt16(Byte[], Int32)
`int`	ToInt32(Byte[], Int32)
`long`	ToInt64(Byte[], Int32)
`float`	ToSingle(Byte[], Int32)
`ushort`	ToUInt16(Byte[], Int32)
`uint`	ToUInt32(Byte[], Int32)
`ulong`	ToUInt64(Byte[], Int32)

Examples

This example initializes an array of bytes, reverses the array if the computer architecture is little-endian (that is, the least significant byte is stored first), and then calls the ToInt32(Byte[], Int32) method to convert four bytes in the array to an `int`. The second argument to ToInt32(Byte[], Int32) specifies the start index of the array of bytes.

> **Note**

The output may differ depending on the endianness of your computer's architecture.

C#

```
byte[] bytes = [0, 0, 0, 25];

// If the system architecture is little-endian (that is, little end first),
// reverse the byte array.
if (BitConverter.IsLittleEndian)
    Array.Reverse(bytes);

int i = BitConverter.ToInt32(bytes, 0);
Console.WriteLine($"int: {i}");
// Output: int: 25
```

In this example, the GetBytes(Int32) method of the BitConverter class is called to convert an `int` to an array of bytes.

> **Note**
>
> The output may differ depending on the endianness of your computer's architecture.

C#

```
byte[] bytes = BitConverter.GetBytes(201805978);
Console.WriteLine("byte array: " + BitConverter.ToString(bytes));
// Output: byte array: 9A-50-07-0C
```

See also

- BitConverter
- IsLittleEndian
- Types

How to convert a string to a number (C# Programming Guide)

You convert a `string` to a number by calling the `Parse` or `TryParse` method found on numeric types (`int`, `long`, `double`, and so on), or by using methods in the System.Convert class.

It's slightly more efficient and straightforward to call a `TryParse` method (for example, int.TryParse("11", out number)) or `Parse` method (for example, var number = int.Parse("11")). Using a Convert method is more useful for general objects that implement IConvertible.

You use `Parse` or `TryParse` methods on the numeric type you expect the string contains, such as the System.Int32 type. The Convert.ToInt32 method uses Parse internally. The `Parse` method returns the converted number; the `TryParse` method returns a boolean value that indicates whether the conversion succeeded, and returns the converted number in an `out` parameter. If the string isn't in a valid format, `Parse` throws an exception, but `TryParse` returns `false`. When calling a `Parse` method, you should always use exception handling to catch a FormatException when the parse operation fails.

> **Tip**
>
> You can use AI assistance to **convert a string to a number**.

Call Parse or TryParse methods

The `Parse` and `TryParse` methods ignore white space at the beginning and at the end of the string, but all other characters must be characters that form the appropriate numeric type (`int`, `long`, `ulong`, `float`, `decimal`, and so on). Any white space within the string that forms the number causes an error. For example, you can use `decimal.TryParse` to parse "10", "10.3", or " 10 ", but you can't use this method to parse 10 from "10X", "1 0" (note the embedded space), "10 .3" (note the embedded space), "10e1" (`float.TryParse` works here), and so on. A string whose value is `null` or String.Empty fails to parse successfully. You can check for a null or empty string before attempting to parse it by calling the String.IsNullOrEmpty method.

The following example demonstrates both successful and unsuccessful calls to `Parse` and `TryParse`.

C#

```
using System;
```

```
public static class StringConversion
{
    public static void Main()
    {
        string input = String.Empty;
        try
        {
            int result = Int32.Parse(input);
            Console.WriteLine(result);
        }
        catch (FormatException)
        {
            Console.WriteLine($"Unable to parse '{input}'");
        }
        // Output: Unable to parse ''

        try
        {
            int numVal = Int32.Parse("-105");
            Console.WriteLine(numVal);
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: -105

        if (Int32.TryParse("-105", out int j))
        {
            Console.WriteLine(j);
        }
        else
        {
            Console.WriteLine("String could not be parsed.");
        }
        // Output: -105

        try
        {
            int m = Int32.Parse("abc");
        }
        catch (FormatException e)
        {
            Console.WriteLine(e.Message);
        }
        // Output: Input string was not in a correct format.

        const string inputString = "abc";
        if (Int32.TryParse(inputString, out int numValue))
        {
            Console.WriteLine(numValue);
        }
        else
        {
            Console.WriteLine($"Int32.TryParse could not parse '{inputString}' to an
```

```
int.");
        }
        // Output: Int32.TryParse could not parse 'abc' to an int.
    }
}
```

The following example illustrates one approach to parsing a string expected to include leading numeric characters (including hexadecimal characters) and trailing non-numeric characters. It assigns valid characters from the beginning of a string to a new string before calling the TryParse method. Because the strings to be parsed contain a few characters, the example calls the String.Concat method to assign valid characters to a new string. For a larger string, the StringBuilder class can be used instead.

C#

```
using System;

public static class StringConversion
{
    public static void Main()
    {
        var str = "  10FFxxx";
        string numericString = string.Empty;
        foreach (var c in str)
        {
            // Check for numeric characters (hex in this case) or leading or
trailing spaces.
            if ((c >= '0' && c <= '9') || (char.ToUpperInvariant(c) >= 'A' &&
char.ToUpperInvariant(c) <= 'F') || c == ' ')
            {
                numericString = string.Concat(numericString, c.ToString());
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, System.Globalization.NumberStyles.HexNumber,
null, out int i))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {i}");
        }
        // Output: '  10FFxxx' --> '  10FF' --> 4351

        str = "   -10FFXXX";
        numericString = "";
        foreach (char c in str)
        {
            // Check for numeric characters (0-9), a negative sign, or leading or
trailing spaces.
            if ((c >= '0' && c <= '9') || c == ' ' || c == '-')
```

```
            {
                numericString = string.Concat(numericString, c);
            }
            else
            {
                break;
            }
        }

        if (int.TryParse(numericString, out int j))
        {
            Console.WriteLine($"'{str}' --> '{numericString}' --> {j}");
        }
        // Output: '   -10FFXXX' --> '   -10' --> -10
    }
}
```

Call Convert methods

The following table lists some of the methods from the Convert class that you can use to convert a string to a number.

Expand table

Numeric type	Method
`decimal`	ToDecimal(String)
`float`	ToSingle(String)
`double`	ToDouble(String)
`short`	ToInt16(String)
`int`	ToInt32(String)
`long`	ToInt64(String)
`ushort`	ToUInt16(String)
`uint`	ToUInt32(String)
`ulong`	ToUInt64(String)

The following example calls the Convert.ToInt32(String) method to convert an input string to an int. The example catches the two most common exceptions thrown by this method: FormatException and OverflowException. If the resulting number can be incremented without exceeding Int32.MaxValue, the example adds 1 to the result and displays the output.

C#

```
using System;

public class ConvertStringExample1
{
    static void Main(string[] args)
    {
        int numVal = -1;
        bool repeat = true;

        while (repeat)
        {
            Console.Write("Enter a number between -2,147,483,648 and +2,147,483,647
(inclusive): ");

            string? input = Console.ReadLine();

            // ToInt32 can throw FormatException or OverflowException.
            try
            {
                numVal = Convert.ToInt32(input);
                if (numVal < Int32.MaxValue)
                {
                    Console.WriteLine($"The new value is {++numVal}");
                }
                else
                {
                    Console.WriteLine("numVal cannot be incremented beyond its
current value");
                }
            }
            catch (FormatException)
            {
                Console.WriteLine("Input string is not a sequence of digits.");
            }
            catch (OverflowException)
            {
                Console.WriteLine("The number cannot fit in an Int32.");
            }

            Console.Write("Go again? Y/N: ");
            string? go = Console.ReadLine();
            if (go?.ToUpper() != "Y")
            {
                repeat = false;
            }
        }
    }
}
// Sample Output:
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): 473
//   The new value is 474
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive):
```

```
2147483647
//   numVal cannot be incremented beyond its current value
//   Go again? Y/N: y
//   Enter a number between -2,147,483,648 and +2,147,483,647 (inclusive): -1000
//   The new value is -999
//   Go again? Y/N: n
```

Use AI to convert a string to a number

You can use AI tools, such as GitHub Copilot, to generate C# code to convert a string to a number. You can customize the prompt to use a string per your requirements.

Here's an example Copilot Chat prompt:

Copilot prompt
`Show me how to parse a string as a number, but don't throw an exception if the input string doesn't represent a number.`

Review Copilot's suggestions before applying them.

For more information, see Copilot FAQs.

See also

- GitHub Copilot in Visual Studio
- GitHub Copilot in VS Code

Last updated on 12/11/2025

How to convert between hexadecimal strings and numeric types (C# Programming Guide)

Article • 10/12/2021

These examples show you how to perform the following tasks:

- Obtain the hexadecimal value of each character in a string.
- Obtain the char that corresponds to each value in a hexadecimal string.
- Convert a hexadecimal `string` to an int.
- Convert a hexadecimal `string` to a float.
- Convert a byte array to a hexadecimal `string`.

Examples

This example outputs the hexadecimal value of each character in a `string`. First it parses the `string` to an array of characters. Then it calls ToInt32(Char) on each character to obtain its numeric value. Finally, it formats the number as its hexadecimal representation in a `string`.

C#

```
string input = "Hello World!";
char[] values = input.ToCharArray();
foreach (char letter in values)
{
    // Get the integral value of the character.
    int value = Convert.ToInt32(letter);
    // Convert the integer value to a hexadecimal value in string form.
    Console.WriteLine($"Hexadecimal value of {letter} is {value:X}");
}
/* Output:
    Hexadecimal value of H is 48
    Hexadecimal value of e is 65
    Hexadecimal value of l is 6C
    Hexadecimal value of l is 6C
    Hexadecimal value of o is 6F
    Hexadecimal value of   is 20
    Hexadecimal value of W is 57
    Hexadecimal value of o is 6F
    Hexadecimal value of r is 72
```

```
    Hexadecimal value of l is 6C
    Hexadecimal value of d is 64
    Hexadecimal value of ! is 21
 */
```

This example parses a `string` of hexadecimal values and outputs the character corresponding to each hexadecimal value. First it calls the Split(Char[]) method to obtain each hexadecimal value as an individual `string` in an array. Then it calls ToInt32(String, Int32) to convert the hexadecimal value to a decimal value represented as an int. It shows two different ways to obtain the character corresponding to that character code. The first technique uses ConvertFromUtf32(Int32), which returns the character corresponding to the integer argument as a `string`. The second technique explicitly casts the `int` to a char.

C#

```
string hexValues = "48 65 6C 6C 6F 20 57 6F 72 6C 64 21";
string[] hexValuesSplit = hexValues.Split(' ');
foreach (string hex in hexValuesSplit)
{
    // Convert the number expressed in base-16 to an integer.
    int value = Convert.ToInt32(hex, 16);
    // Get the character corresponding to the integral value.
    string stringValue = Char.ConvertFromUtf32(value);
    char charValue = (char)value;
    Console.WriteLine("hexadecimal value = {0}, int value = {1}, char value
= {2} or {3}",
                      hex, value, stringValue, charValue);
}
/* Output:
    hexadecimal value = 48, int value = 72, char value = H or H
    hexadecimal value = 65, int value = 101, char value = e or e
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 20, int value = 32, char value =   or
    hexadecimal value = 57, int value = 87, char value = W or W
    hexadecimal value = 6F, int value = 111, char value = o or o
    hexadecimal value = 72, int value = 114, char value = r or r
    hexadecimal value = 6C, int value = 108, char value = l or l
    hexadecimal value = 64, int value = 100, char value = d or d
    hexadecimal value = 21, int value = 33, char value = ! or !
*/
```

This example shows another way to convert a hexadecimal `string` to an integer, by calling the Parse(String, NumberStyles) method.

C#

```
string hexString = "8E2";
int num = Int32.Parse(hexString,
System.Globalization.NumberStyles.HexNumber);
Console.WriteLine(num);
//Output: 2274
```

The following example shows how to convert a hexadecimal `string` to a float by using the System.BitConverter class and the UInt32.Parse method.

C#

```
string hexString = "43480170";
uint num = uint.Parse(hexString,
System.Globalization.NumberStyles.AllowHexSpecifier);

byte[] floatVals = BitConverter.GetBytes(num);
float f = BitConverter.ToSingle(floatVals, 0);
Console.WriteLine($"float convert = {f}");

// Output: 200.0056
```

The following example shows how to convert a byte array to a hexadecimal string by using the System.BitConverter class.

C#

```
byte[] vals = [0x01, 0xAA, 0xB1, 0xDC, 0x10, 0xDD];

string str = BitConverter.ToString(vals);
Console.WriteLine(str);

str = BitConverter.ToString(vals).Replace("-", "");
Console.WriteLine(str);

/*Output:
  01-AA-B1-DC-10-DD
  01AAB1DC10DD
 */
```

The following example shows how to convert a byte array to a hexadecimal string by calling the Convert.ToHexString method introduced in .NET 5.0.

C#

```
byte[] array = [0x64, 0x6f, 0x74, 0x63, 0x65, 0x74];

string hexValue = Convert.ToHexString(array);
```

```
Console.WriteLine(hexValue);

/*Output:
  646F74636574
 */
```

See also

- Standard Numeric Format Strings
- Types
- How to determine whether a string represents a numeric value

Versioning with the Override and New Keywords (C# Programming Guide)

Article • 10/27/2021

The C# language is designed so that versioning between base and derived classes in different libraries can evolve and maintain backward compatibility. This means, for example, that the introduction of a new member in a base class with the same name as a member in a derived class is completely supported by C# and does not lead to unexpected behavior. It also means that a class must explicitly state whether a method is intended to override an inherited method, or whether a method is a new method that hides a similarly named inherited method.

In C#, derived classes can contain methods with the same name as base class methods.

- If the method in the derived class is not preceded by new or override keywords, the compiler will issue a warning and the method will behave as if the `new` keyword were present.
- If the method in the derived class is preceded with the `new` keyword, the method is defined as being independent of the method in the base class.
- If the method in the derived class is preceded with the `override` keyword, objects of the derived class will call that method instead of the base class method.
- In order to apply the `override` keyword to the method in the derived class, the base class method must be defined virtual.
- The base class method can be called from within the derived class using the `base` keyword.
- The `override`, `virtual`, and `new` keywords can also be applied to properties, indexers, and events.

By default, C# methods are not virtual. If a method is declared as virtual, any class inheriting the method can implement its own version. To make a method virtual, the `virtual` modifier is used in the method declaration of the base class. The derived class can then override the base virtual method by using the `override` keyword or hide the virtual method in the base class by using the `new` keyword. If neither the `override` keyword nor the `new` keyword is specified, the compiler will issue a warning and the method in the derived class will hide the method in the base class.

To demonstrate this in practice, assume for a moment that Company A has created a class named `GraphicsClass`, which your program uses. The following is `GraphicsClass`:

```
C#

class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
}
```

Your company uses this class, and you use it to derive your own class, adding a new method:

```
C#

class YourDerivedGraphicsClass : GraphicsClass
{
    public void DrawRectangle() { }
}
```

Your application is used without problems, until Company A releases a new version of `GraphicsClass`, which resembles the following code:

```
C#

class GraphicsClass
{
    public virtual void DrawLine() { }
    public virtual void DrawPoint() { }
    public virtual void DrawRectangle() { }
}
```

The new version of `GraphicsClass` now contains a method named `DrawRectangle`. Initially, nothing occurs. The new version is still binary compatible with the old version. Any software that you have deployed will continue to work, even if the new class is installed on those computer systems. Any existing calls to the method `DrawRectangle` will continue to reference your version, in your derived class.

However, as soon as you recompile your application by using the new version of `GraphicsClass`, you will receive a warning from the compiler, CS0108. This warning informs you that you have to consider how you want your `DrawRectangle` method to behave in your application.

If you want your method to override the new base class method, use the `override` keyword:

```
C#

class YourDerivedGraphicsClass : GraphicsClass
{
    public override void DrawRectangle() { }
}
```

The `override` keyword makes sure that any objects derived from `YourDerivedGraphicsClass` will use the derived class version of `DrawRectangle`. Objects derived from `YourDerivedGraphicsClass` can still access the base class version of `DrawRectangle` by using the base keyword:

```
C#

base.DrawRectangle();
```

If you do not want your method to override the new base class method, the following considerations apply. To avoid confusion between the two methods, you can rename your method. This can be time-consuming and error-prone, and just not practical in some cases. However, if your project is relatively small, you can use Visual Studio's Refactoring options to rename the method. For more information, see Refactoring Classes and Types (Class Designer).

Alternatively, you can prevent the warning by using the keyword `new` in your derived class definition:

```
C#

class YourDerivedGraphicsClass : GraphicsClass
{
    public new void DrawRectangle() { }
}
```

Using the `new` keyword tells the compiler that your definition hides the definition that is contained in the base class. This is the default behavior.

Override and Method Selection

When a method is named on a class, the C# compiler selects the best method to call if more than one method is compatible with the call, such as when there are two methods

with the same name, and parameters that are compatible with the parameter passed. The following methods would be compatible:

```
C#
```

```
public class Derived : Base
{
    public override void DoWork(int param) { }
    public void DoWork(double param) { }
}
```

When `DoWork` is called on an instance of `Derived`, the C# compiler will first try to make the call compatible with the versions of `DoWork` declared originally on `Derived`. Override methods are not considered as declared on a class, they are new implementations of a method declared on a base class. Only if the C# compiler cannot match the method call to an original method on `Derived`, it will try to match the call to an overridden method with the same name and compatible parameters. For example:

```
C#
```

```
int val = 5;
Derived d = new();
d.DoWork(val);  // Calls DoWork(double).
```

Because the variable `val` can be converted to a double implicitly, the C# compiler calls `DoWork(double)` instead of `DoWork(int)`. There are two ways to avoid this. First, avoid declaring new methods with the same name as virtual methods. Second, you can instruct the C# compiler to call the virtual method by making it search the base class method list by casting the instance of `Derived` to `Base`. Because the method is virtual, the implementation of `DoWork(int)` on `Derived` will be called. For example:

```
C#
```

```
((Base)d).DoWork(val);  // Calls DoWork(int) on Derived.
```

For more examples of `new` and `override`, see Knowing When to Use Override and New Keywords.

See also

- The C# type system
- Methods

- Inheritance

Knowing When to Use Override and New Keywords (C# Programming Guide)

08/02/2025

In C#, a method in a derived class can have the same name as a method in the base class. You can specify how the methods interact by using the new and override keywords. The `override` modifier *extends* the base class `virtual` method, and the `new` modifier *hides* an accessible base class method. The difference is illustrated in the examples in this topic.

In a console application, declare the following two classes, `BaseClass` and `DerivedClass`. `DerivedClass` inherits from `BaseClass`.

C#

```
class BaseClass
{
    public void Method1()
    {
        Console.WriteLine("Base - Method1");
    }
}

class DerivedClass : BaseClass
{
    public void Method2()
    {
        Console.WriteLine("Derived - Method2");
    }
}
```

In the `Main` method, declare variables `bc`, `dc`, and `bcdc`.

- `bc` is of type `BaseClass`, and its value is of type `BaseClass`.
- `dc` is of type `DerivedClass`, and its value is of type `DerivedClass`.
- `bcdc` is of type `BaseClass`, and its value is of type `DerivedClass`. This is the variable to pay attention to.

Because `bc` and `bcdc` have type `BaseClass`, they can only directly access `Method1`, unless you use casting. Variable `dc` can access both `Method1` and `Method2`. These relationships are shown in the following code.

C#

```
class Program
{
    static void Main(string[] args)
    {
        BaseClass bc = new BaseClass();
        DerivedClass dc = new DerivedClass();
        BaseClass bcdc = new DerivedClass();

        bc.Method1();
        dc.Method1();
        dc.Method2();
        bcdc.Method1();
    }
    // Output:
    // Base - Method1
    // Base - Method1
    // Derived - Method2
    // Base - Method1
}
```

Next, add the following `Method2` method to `BaseClass`. The signature of this method matches the signature of the `Method2` method in `DerivedClass`.

C#

```
public void Method2()
{
    Console.WriteLine("Base - Method2");
}
```

Because `BaseClass` now has a `Method2` method, a second calling statement can be added for `BaseClass` variables `bc` and `bcdc`, as shown in the following code.

C#

```
bc.Method1();
bc.Method2();
dc.Method1();
dc.Method2();
bcdc.Method1();
bcdc.Method2();
```

The New Keyword

When you build the project, you see that the addition of the `Method2` method in `BaseClass` causes a warning. The warning says that the `Method2` method in `DerivedClass` hides the

`Method2` method in `BaseClass`. You are advised to use the `new` keyword in the `Method2` definition if you intend to cause that result. Alternatively, you could rename one of the `Method2` methods to resolve the warning, but that is not always practical.

Before adding `new`, run the program to see the output produced by the additional calling statements. The following results are displayed.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Base - Method1
// Derived - Method2
// Base - Method1
// Base - Method2
```

The `new` keyword preserves the relationships that produce that output, but it suppresses the warning. The variables that have type `BaseClass` continue to access the members of `BaseClass`, and the variable that has type `DerivedClass` continues to access members in `DerivedClass` first, and then to consider members inherited from `BaseClass`.

To suppress the warning, add the `new` modifier to the definition of `Method2` in `DerivedClass`, as shown in the following code. The modifier can be added before or after `public`.

C#

```
public new void Method2()
{
    Console.WriteLine("Derived - Method2");
}
```

Run the program again to verify that the output has not changed. Also verify that the warning no longer appears. By using `new`, you are asserting that you are aware that the member that it modifies hides a member that is inherited from the base class. For more information about name hiding through inheritance, see new Modifier.

Virtual and Override Keywords

To contrast this behavior to the effects of using `override`, add the following method to `DerivedClass`. The `override` modifier can be added before or after `public`.

C#

```
public override void Method1()
{
    Console.WriteLine("Derived - Method1");
}
```

Add the `virtual` modifier to the definition of `Method1` in `BaseClass`. The `virtual` modifier can be added before or after `public`.

C#

```
public virtual void Method1()
{
    Console.WriteLine("Base - Method1");
}
```

Run the project again. Notice especially the last two lines of the following output.

C#

```
// Output:
// Base - Method1
// Base - Method2
// Derived - Method1
// Derived - Method2
// Derived - Method1
// Base - Method2
```

The use of the `override` modifier enables `bcdc` to access the `Method1` method that is defined in `DerivedClass`. Typically, that is the desired behavior in inheritance hierarchies. You want objects that have values that are created from the derived class to use the methods that are defined in the derived class. You achieve that behavior by using `override` to extend the base class method.

The following code contains the full example.

C#

```
using System;
using System.Text;

namespace OverrideAndNew
{
    class Program
    {
        static void Main(string[] args)
        {
            BaseClass bc = new BaseClass();
```

```
            DerivedClass dc = new DerivedClass();
            BaseClass bcdc = new DerivedClass();

            // The following two calls do what you would expect. They call
            // the methods that are defined in BaseClass.
            bc.Method1();
            bc.Method2();
            // Output:
            // Base - Method1
            // Base - Method2

            // The following two calls do what you would expect. They call
            // the methods that are defined in DerivedClass.
            dc.Method1();
            dc.Method2();
            // Output:
            // Derived - Method1
            // Derived - Method2

            // The following two calls produce different results, depending
            // on whether override (Method1) or new (Method2) is used.
            bcdc.Method1();
            bcdc.Method2();
            // Output:
            // Derived - Method1
            // Base - Method2
        }
    }

    class BaseClass
    {
        public virtual void Method1()
        {
            Console.WriteLine("Base - Method1");
        }

        public virtual void Method2()
        {
            Console.WriteLine("Base - Method2");
        }
    }

    class DerivedClass : BaseClass
    {
        public override void Method1()
        {
            Console.WriteLine("Derived - Method1");
        }

        public new void Method2()
        {
            Console.WriteLine("Derived - Method2");
        }
    }
}
```

Override and New in Derived Classes

The following example illustrates similar behavior in a different context. The example defines three classes: a base class named `Car` and two classes that are derived from it, `ConvertibleCar` and `Minivan`. The base class contains a `DescribeCar` method. The method displays a basic description of a car, and then calls `ShowDetails` to provide additional information. Each of the three classes defines a `ShowDetails` method. The `new` modifier is used to define `ShowDetails` in the `ConvertibleCar` class. The `override` modifier is used to define `ShowDetails` in the `Minivan` class.

C#

```
// Define the base class, Car. The class defines two methods,
// DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each derived
// class also defines a ShowDetails method. The example tests which version of
// ShowDetails is selected, the base class method or the derived class method.
class Car
{
    public void DescribeCar()
    {
        System.Console.WriteLine("Four wheels and an engine.");
        ShowDetails();
    }

    public virtual void ShowDetails()
    {
        System.Console.WriteLine("Standard transportation.");
    }
}

// Define the derived classes.

// Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
// hides the base class method.
class ConvertibleCar : Car
{
    public new void ShowDetails()
    {
        System.Console.WriteLine("A roof that opens up.");
    }
}

// Class Minivan uses the override modifier to specify that ShowDetails
// extends the base class method.
class Minivan : Car
{
    public override void ShowDetails()
    {
```

```
        System.Console.WriteLine("Carries seven people.");
    }
}
```

The example tests which version of `ShowDetails` is called. The following method, `TestCars1`, declares an instance of each class, and then calls `DescribeCar` on each instance.

C#

```
public static void TestCars1()
{
    System.Console.WriteLine("\nTestCars1");
    System.Console.WriteLine("----------");

    Car car1 = new Car();
    car1.DescribeCar();
    System.Console.WriteLine("----------");

    // Notice the output from this test case. The new modifier is
    // used in the definition of ShowDetails in the ConvertibleCar
    // class.

    ConvertibleCar car2 = new ConvertibleCar();
    car2.DescribeCar();
    System.Console.WriteLine("----------");

    Minivan car3 = new Minivan();
    car3.DescribeCar();
    System.Console.WriteLine("----------");
}
```

`TestCars1` produces the following output. Notice especially the results for `car2`, which probably are not what you expected. The type of the object is `ConvertibleCar`, but `DescribeCar` does not access the version of `ShowDetails` that is defined in the `ConvertibleCar` class because that method is declared with the `new` modifier, not the `override` modifier. As a result, a `ConvertibleCar` object displays the same description as a `Car` object. Contrast the results for `car3`, which is a `Minivan` object. In this case, the `ShowDetails` method that is declared in the `Minivan` class overrides the `ShowDetails` method that is declared in the `Car` class, and the description that is displayed describes a minivan.

C#

```
// TestCars1
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
```

```
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

`TestCars2` creates a list of objects that have type `Car`. The values of the objects are instantiated from the `Car`, `ConvertibleCar`, and `Minivan` classes. `DescribeCar` is called on each element of the list. The following code shows the definition of `TestCars2`.

C#

```
public static void TestCars2()
{
    System.Console.WriteLine("\nTestCars2");
    System.Console.WriteLine("----------");

    var cars = new List<Car> { new Car(), new ConvertibleCar(),
        new Minivan() };

    foreach (var car in cars)
    {
        car.DescribeCar();
        System.Console.WriteLine("----------");
    }
}
```

The following output is displayed. Notice that it is the same as the output that is displayed by `TestCars1`. The `ShowDetails` method of the `ConvertibleCar` class is not called, regardless of whether the type of the object is `ConvertibleCar`, as in `TestCars1`, or `Car`, as in `TestCars2`. Conversely, `car3` calls the `ShowDetails` method from the `Minivan` class in both cases, whether it has type `Minivan` or type `Car`.

C#

```
// TestCars2
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Standard transportation.
// ----------
// Four wheels and an engine.
// Carries seven people.
// ----------
```

Methods `TestCars3` and `TestCars4` complete the example. These methods call `ShowDetails` directly, first from objects declared to have type `ConvertibleCar` and `Minivan` (`TestCars3`), then from objects declared to have type `Car` (`TestCars4`). The following code defines these two methods.

C#

```
public static void TestCars3()
{
    System.Console.WriteLine("\nTestCars3");
    System.Console.WriteLine("----------");
    ConvertibleCar car2 = new ConvertibleCar();
    Minivan car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}

public static void TestCars4()
{
    System.Console.WriteLine("\nTestCars4");
    System.Console.WriteLine("----------");
    Car car2 = new ConvertibleCar();
    Car car3 = new Minivan();
    car2.ShowDetails();
    car3.ShowDetails();
}
```

The methods produce the following output, which corresponds to the results from the first example in this topic.

C#

```
// TestCars3
// ----------
// A roof that opens up.
// Carries seven people.

// TestCars4
// ----------
// Standard transportation.
// Carries seven people.
```

The following code shows the complete project and its output.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
```

```
using System.Text;

namespace OverrideAndNew2
{
    class Program
    {
        static void Main(string[] args)
        {
            // Declare objects of the derived classes and test which version
            // of ShowDetails is run, base or derived.
            TestCars1();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars2();

            // Declare objects of the derived classes and call ShowDetails
            // directly.
            TestCars3();

            // Declare objects of the base class, instantiated with the
            // derived classes, and repeat the tests.
            TestCars4();
        }

        public static void TestCars1()
        {
            System.Console.WriteLine("\nTestCars1");
            System.Console.WriteLine("----------");

            Car car1 = new Car();
            car1.DescribeCar();
            System.Console.WriteLine("----------");

            // Notice the output from this test case. The new modifier is
            // used in the definition of ShowDetails in the ConvertibleCar
            // class.
            ConvertibleCar car2 = new ConvertibleCar();
            car2.DescribeCar();
            System.Console.WriteLine("----------");

            Minivan car3 = new Minivan();
            car3.DescribeCar();
            System.Console.WriteLine("----------");
        }
        // Output:
        // TestCars1
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
```

```
        // Carries seven people.
        // ----------

        public static void TestCars2()
        {
            System.Console.WriteLine("\nTestCars2");
            System.Console.WriteLine("----------");

            var cars = new List<Car> { new Car(), new ConvertibleCar(),
                new Minivan() };

            foreach (var car in cars)
            {
                car.DescribeCar();
                System.Console.WriteLine("----------");
            }
        }
        // Output:
        // TestCars2
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Standard transportation.
        // ----------
        // Four wheels and an engine.
        // Carries seven people.
        // ----------

        public static void TestCars3()
        {
            System.Console.WriteLine("\nTestCars3");
            System.Console.WriteLine("----------");
            ConvertibleCar car2 = new ConvertibleCar();
            Minivan car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
        // TestCars3
        // ----------
        // A roof that opens up.
        // Carries seven people.

        public static void TestCars4()
        {
            System.Console.WriteLine("\nTestCars4");
            System.Console.WriteLine("----------");
            Car car2 = new ConvertibleCar();
            Car car3 = new Minivan();
            car2.ShowDetails();
            car3.ShowDetails();
        }
        // Output:
```

```
        // TestCars4
        // ----------
        // Standard transportation.
        // Carries seven people.
    }

    // Define the base class, Car. The class defines two virtual methods,
    // DescribeCar and ShowDetails. DescribeCar calls ShowDetails, and each
derived
    // class also defines a ShowDetails method. The example tests which version of
    // ShowDetails is used, the base class method or the derived class method.
    class Car
    {
        public virtual void DescribeCar()
        {
            System.Console.WriteLine("Four wheels and an engine.");
            ShowDetails();
        }

        public virtual void ShowDetails()
        {
            System.Console.WriteLine("Standard transportation.");
        }
    }

    // Define the derived classes.

    // Class ConvertibleCar uses the new modifier to acknowledge that ShowDetails
    // hides the base class method.
    class ConvertibleCar : Car
    {
        public new void ShowDetails()
        {
            System.Console.WriteLine("A roof that opens up.");
        }
    }

    // Class Minivan uses the override modifier to specify that ShowDetails
    // extends the base class method.
    class Minivan : Car
    {
        public override void ShowDetails()
        {
            System.Console.WriteLine("Carries seven people.");
        }
    }

}
```

See also

- The C# type system

- Versioning with the Override and New Keywords
- base
- abstract

How to override the ToString method (C# Programming Guide)

Article • 10/27/2021

Every class or struct in C# implicitly inherits the Object class. Therefore, every object in C# gets the ToString method, which returns a string representation of that object. For example, all variables of type `int` have a `ToString` method, which enables them to return their contents as a string:

```
C#

int x = 42;
string strx = x.ToString();
Console.WriteLine(strx);
// Output:
// 42
```

When you create a custom class or struct, you should override the ToString method in order to provide information about your type to client code.

For information about how to use format strings and other types of custom formatting with the `ToString` method, see Formatting Types.

> **Important**
>
> When you decide what information to provide through this method, consider whether your class or struct will ever be used by untrusted code. Be careful to ensure that you do not provide any information that could be exploited by malicious code.

To override the `ToString` method in your class or struct:

1. Declare a `ToString` method with the following modifiers and return type:

   ```
   C#

   public override string ToString(){}
   ```

2. Implement the method so that it returns a string.

The following example returns the name of the class in addition to the data specific to a particular instance of the class.

C#

```
class Person
{
    public string Name { get; set; }
    public int Age { get; set; }

    public override string ToString()
    {
        return "Person: " + Name + " " + Age;
    }
}
```

You can test the `ToString` method as shown in the following code example:

C#

```
Person person = new() { Name = "John", Age = 12 };
Console.WriteLine(person);
// Output:
// Person: John 12
```

See also

- IFormattable
- The C# type system
- Strings
- string
- override
- virtual
- Formatting Types

Members (C# Programming Guide)

Article • 09/17/2021

Classes and structs have members that represent their data and behavior. A class's members include all the members declared in the class, along with all members (except constructors and finalizers) declared in all classes in its inheritance hierarchy. Private members in base classes are inherited but are not accessible from derived classes.

The following table lists the kinds of members a class or struct may contain:

Expand table

Member	Description
Fields	Fields are variables declared at class scope. A field may be a built-in numeric type or an instance of another class. For example, a calendar class may have a field that contains the current date.
Constants	Constants are fields whose value is set at compile time and cannot be changed.
Properties	Properties are methods on a class that are accessed as if they were fields on that class. A property can provide protection for a class field to keep it from being changed without the knowledge of the object.
Methods	Methods define the actions that a class can perform. Methods can take parameters that provide input data, and can return output data through parameters. Methods can also return a value directly, without using a parameter.
Events	Events provide notifications about occurrences, such as button clicks or the successful completion of a method, to other objects. Events are defined and triggered by using delegates.
Operators	Overloaded operators are considered type members. When you overload an operator, you define it as a public static method in a type. For more information, see Operator overloading.
Indexers	Indexers enable an object to be indexed in a manner similar to arrays.
Constructors	Constructors are methods that are called when the object is first created. They are often used to initialize the data of an object.
Finalizers	Finalizers are used very rarely in C#. They are methods that are called by the runtime execution engine when the object is about to be removed from memory. They are generally used to make sure that any resources which must be released are handled appropriately.
Nested Types	Nested types are types declared within another type. Nested types are often used to describe objects that are used only by the types that contain them.

See also

- Classes

Abstract and Sealed Classes and Class Members (C# Programming Guide)

Article • 10/27/2021

The abstract keyword enables you to create classes and class members that are incomplete and must be implemented in a derived class.

The sealed keyword enables you to prevent the inheritance of a class or certain class members that were previously marked virtual.

Abstract Classes and Class Members

Classes can be declared as abstract by putting the keyword `abstract` before the class definition. For example:

C#

```
public abstract class A
{
    // Class members here.
}
```

An abstract class cannot be instantiated. The purpose of an abstract class is to provide a common definition of a base class that multiple derived classes can share. For example, a class library may define an abstract class that is used as a parameter to many of its functions, and require programmers using that library to provide their own implementation of the class by creating a derived class.

Abstract classes may also define abstract methods. This is accomplished by adding the keyword `abstract` before the return type of the method. For example:

C#

```
public abstract class A
{
    public abstract void DoWork(int i);
}
```

Abstract methods have no implementation, so the method definition is followed by a semicolon instead of a normal method block. Derived classes of the abstract class must implement all abstract methods. When an abstract class inherits a virtual method from a

base class, the abstract class can override the virtual method with an abstract method. For example:

```C#
// compile with: -target:library
public class D
{
    public virtual void DoWork(int i)
    {
        // Original implementation.
    }
}

public abstract class E : D
{
    public abstract override void DoWork(int i);
}

public class F : E
{
    public override void DoWork(int i)
    {
        // New implementation.
    }
}
```

If a `virtual` method is declared `abstract`, it is still virtual to any class inheriting from the abstract class. A class inheriting an abstract method cannot access the original implementation of the method—in the previous example, `DoWork` on class F cannot call `DoWork` on class D. In this way, an abstract class can force derived classes to provide new method implementations for virtual methods.

Sealed Classes and Class Members

Classes can be declared as sealed by putting the keyword `sealed` before the class definition. For example:

```C#
public sealed class D
{
    // Class members here.
}
```

A sealed class cannot be used as a base class. For this reason, it cannot also be an abstract class. Sealed classes prevent derivation. Because they can never be used as a

base class, some run-time optimizations can make calling sealed class members slightly faster.

A method, indexer, property, or event, on a derived class that is overriding a virtual member of the base class can declare that member as sealed. This negates the virtual aspect of the member for any further derived class. This is accomplished by putting the `sealed` keyword before the override keyword in the class member declaration. For example:

```
C#

public class D : C
{
    public sealed override void DoWork() { }
}
```

See also

- The C# type system
- Inheritance
- Methods
- Fields
- How to define abstract properties

Static Classes and Static Class Members (C# Programming Guide)

10/17/2025

A static class is basically the same as a non-static class, but there's one difference: a static class can't be instantiated. In other words, you can't use the new operator to create a variable of the class type. Because there's no instance variable, you access the members of a static class by using the class name itself. For example, if you have a static class that is named `UtilityClass` that has a public static method named `MethodA`, you call the method as shown in the following example:

C#

```
UtilityClass.MethodA();
```

A static class can be used as a convenient container for sets of methods that just operate on input parameters and don't have to get or set any internal instance fields. For example, in the .NET Class Library, the static System.Math class contains methods that perform mathematical operations, without any requirement to store or retrieve data that's unique to a particular instance of the Math class. That is, you apply the members of the class by specifying the class name and the method name, as shown in the following example.

C#

```
double dub = -3.14;
Console.WriteLine(Math.Abs(dub));
Console.WriteLine(Math.Floor(dub));
Console.WriteLine(Math.Round(Math.Abs(dub)));

// Output:
// 3.14
// -4
// 3
```

As is the case with all class types, the .NET runtime loads the type information for a static class when the program that references the class is loaded. The program can't specify exactly when the class is loaded. However, it's guaranteed to load and have its fields initialized and its static constructor called before the class is referenced for the first time in your program. A static constructor is only called one time, and a static class remains in memory for the lifetime of the application domain in which your program resides.

> **Note**

To create a non-static class that allows only one instance of itself to be created, see Implementing Singleton in C#.

The following list provides the main features of a static class:

- Contains only static members.
- Can't be instantiated.
- Is sealed.
- Can't contain Instance Constructors.

Creating a static class is therefore basically the same as creating a class that contains only static members and a private constructor. A private constructor prevents the class from being instantiated. The advantage of using a static class is that the compiler can check to make sure that no instance members are accidentally added. The compiler guarantees that instances of this class can't be created.

Static classes are sealed and therefore can't be inherited. They can't inherit from any class or interface except Object. Static classes can't contain an instance constructor. However, they can contain a static constructor. Non-static classes should also define a static constructor if the class contains static members that require non-trivial initialization. For more information, see Static Constructors.

Example

Here's an example of a static class that contains two methods that convert temperature from Celsius to Fahrenheit and from Fahrenheit to Celsius:

C#

```
public static class TemperatureConverter
{
    public static double CelsiusToFahrenheit(string temperatureCelsius) =>
        double.Parse(temperatureCelsius) * 9 / 5 + 32;

    public static double FahrenheitToCelsius(string temperatureFahrenheit) =>
        (double.Parse(temperatureFahrenheit) - 32) * 5 / 9;
}

class TestTemperatureConverter
{
    static void Main()
    {
        Console.WriteLine("Please select the convertor direction");
```

```
        Console.WriteLine("1. From Celsius to Fahrenheit.");
        Console.WriteLine("2. From Fahrenheit to Celsius.");
        Console.Write(":");

        string? selection = Console.ReadLine();

        switch (selection)
        {
            case "1":
                Console.Write("Please enter the Celsius temperature: ");
                var f =
TemperatureConverter.CelsiusToFahrenheit(Console.ReadLine() ?? "0");
                Console.WriteLine($"Temperature in Fahrenheit: {f:F2}");
                break;

            case "2":
                Console.Write("Please enter the Fahrenheit temperature: ");
                var c =
TemperatureConverter.FahrenheitToCelsius(Console.ReadLine() ?? "0");
                Console.WriteLine($"Temperature in Celsius: {c:F2}");
                break;

            default:
                Console.WriteLine("Please select a convertor.");
                break;
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Example Output:
    Please select the convertor direction
    1. From Celsius to Fahrenheit.
    2. From Fahrenheit to Celsius.
    :2
    Please enter the Fahrenheit temperature: 20
    Temperature in Celsius: -6.67
    Press any key to exit.
 */
```

Static Members

A non-static class can contain static methods, fields, properties, or events. The static member is callable on a class even when no instance of the class exists. The static member is always accessed by the class name, not the instance name. Only one copy of a static member exists, regardless of how many instances of the class are created. For generic types, each closed generic type has its own copy of static members. Static fields marked with ThreadStaticAttribute have one copy per thread. Static methods and properties can't access

non-static fields and events in their containing type, and they can't access an instance variable of any object unless it's explicitly passed in a method parameter.

It's more typical to declare a non-static class with some static members, than to declare an entire class as static. Two common uses of static fields are to keep a count of the number of objects that are instantiated, or to store a value that must be shared among all instances.

Static methods can be overloaded but not overridden, because they belong to the class, and not to any instance of the class.

Although a field can't be declared as `static const`, a const field is essentially static in its behavior. It belongs to the type, not to instances of the type. Therefore, `const` fields can be accessed by using the same `ClassName.MemberName` notation used for static fields. No object instance is required.

C# doesn't support static local variables (that is, variables that are declared in method scope).

You declare static class members by using the `static` keyword before the return type of the member, as shown in the following example:

C#

```
public class Automobile
{
    public static int NumberOfWheels = 4;

    public static int SizeOfGasTank => 15;

    public static void Drive() { }

    public static event EventType? RunOutOfGas;

    // Other non-static fields and properties...
}
```

Static members are initialized before the static member is accessed for the first time and before the static constructor, if there's one, is called. To access a static class member, use the name of the class instead of a variable name to specify the location of the member, as shown in the following example:

C#

```
Automobile.Drive();
var i = Automobile.NumberOfWheels;
```

If your class contains static fields, provide a static constructor that initializes them when the class is loaded.

A call to a static method generates a call instruction in common intermediate language (CIL), whereas a call to an instance method generates a `callvirt` instruction, which also checks for null object references. However, most of the time the performance difference between the two isn't significant.

C# Language Specification

For more information, see Static classes, Static and instance members and Static constructors in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- static
- Classes
- class
- Static Constructors
- Instance Constructors

Access modifiers (C# programming guide)

All types and type members have an accessibility level. The accessibility level controls whether other code in your assembly or other assemblies can use them. An assembly is a *.dll* or *.exe* created by compiling one or more *.cs* files in a single compilation. Use the following access modifiers to specify the accessibility of a type or member when you declare it:

- public: Code in any assembly can access this type or member. The accessibility level of the containing type controls the accessibility level of public members of the type.
- private: Only code declared in the same `class` or `struct` can access this member.
- protected: Only code in the same `class` or in a derived `class` can access this type or member.
- internal: Only code in the same assembly can access this type or member.
- protected internal: Only code in the same assembly *or* in a derived class in another assembly can access this type or member.
- private protected: Only code in the same assembly *and* in the same class or a derived class can access the type or member.

Use the file modifier for top-level (non-nested) types. The `file` modifier restricts access to code in the same source file. You can't combine the `file` modifier with any access modifier.

The record modifier on a type causes the compiler to synthesize extra members. The `record` modifier doesn't affect the default accessibility for either a `record class` or a `record struct`.

Summary table

Expand table

Caller's location	`public`	`protected internal`	`protected`	`internal`	`private protected`	`private`
Within the class	✔	✔	✔	✔	✔	✔
Derived class (same assembly)	✔	✔	✔	✔	✔	✘
Non-derived class (same assembly)	✔	✔	✘	✔	✘	✘
Derived class (different assembly)	✔	✔	✔	✘	✘	✘
Non-derived class (different assembly)	✔	✘	✘	✘	✘	✘

The following examples demonstrate how to specify access modifiers on a type and member:

C#

```
public class Bicycle
{
    public void Pedal() { }
}
```

Not all access modifiers are valid for all types or members in all contexts. In some cases, the accessibility of the containing type constrains the accessibility of its members.

Multiple declarations of a partial class or partial member must have the same accessibility. If one declaration doesn't include an access modifier, the other declarations can't declare an access modifier. The compiler generates an error if multiple declarations for the partial class or method declare different accessibilities.

Class and struct accessibility

Classes and structs declared directly within a namespace (aren't nested within other classes or structs) can have `public` or `internal` access. If you don't specify an access modifier, the default is `internal`. You can also use the `file` modifier to restrict access to the current source file.

You can declare struct members, including nested classes and structs, as `public`, `internal`, or `private`. You can declare class members, including nested classes and structs, as `public`, `protected internal`, `protected`, `internal`, `private protected`, or `private`. By default, class and struct members, including nested classes and structs, have `private` access.

Derived classes can't have greater accessibility than their base types. You can't declare a public class `B` that derives from an internal class `A`. If allowed, it would have the effect of making `A` public, because all `protected` or `internal` members of `A` are accessible from the derived class.

You can enable specific other assemblies to access your internal types by using the `InternalsVisibleToAttribute`. For more information, see Friend Assemblies.

Other types

You can declare interfaces directly within a namespace as `public` or `internal`. Just like classes and structs, interfaces default to `internal` access. Interface members are `public` by default because the purpose of an interface is to enable other types to access a class or struct. Interface member declarations might include any access modifier. Use access modifiers on

`interface` members to provide a common implementation needed by all implementors of an interface.

A delegate type declared directly in a namespace has `internal` access by default.

For more information about access modifiers, see the Accessibility Levels page.

Member accessibility

You can declare members of a `class` or `struct` (including nested classes and structs) with any of the six types of access. You can't declare struct members as `protected`, `protected internal`, or `private protected` because structs don't support inheritance.

Normally, the accessibility of a member isn't greater than the accessibility of the type that contains it. However, you can access a `public` member of an `internal` class from outside the assembly if the member implements interface methods or overrides virtual methods that are defined in a public base class.

The type of any member field, property, or event must be at least as accessible as the member itself. Similarly, the return type and the parameter types of any method, indexer, or delegate must be at least as accessible as the member itself. For example, you can't have a `public` method `M` that returns a class `C` unless `C` is also `public`. Likewise, you can't have a `protected` property of type `A` if `A` is declared as `private`.

You must always declare user-defined operators as `public`. For more information, see Operator overloading.

To set the access level for a `class` or `struct` member, add the appropriate keyword to the member declaration, as shown in the following example.

C#

```
// public class:
public class Tricycle
{
    // protected method:
    protected void Pedal() { }

    // private field:
    private int _wheels = 3;

    // protected internal property:
    protected internal int Wheels => _wheels;
}
```

Finalizers can't have accessibility modifiers. Members of an `enum` type are always `public`, and you can't apply access modifiers.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Specify modifier order (style rule IDE0036)
- The C# type system
- Interfaces
- Accessibility Levels
- private
- public
- internal
- protected
- protected internal
- private protected
- sealed
- class
- struct
- interface
- Anonymous types

Last updated on 12/17/2025

Fields (C# Programming Guide)

Article • 05/26/2023

A *field* is a variable of any type that is declared directly in a class or struct. Fields are *members* of their containing type.

A class or struct may have instance fields, static fields, or both. Instance fields are specific to an instance of a type. If you have a class `T`, with an instance field `F`, you can create two objects of type `T`, and modify the value of `F` in each object without affecting the value in the other object. By contrast, a static field belongs to the type itself, and is shared among all instances of that type. You can access the static field only by using the type name. If you access the static field by an instance name, you get CS0176 compile-time error.

Generally, you should declare `private` or `protected` accessibility for fields. Data that your type exposes to client code should be provided through methods, properties, and indexers. By using these constructs for indirect access to internal fields, you can guard against invalid input values. A private field that stores the data exposed by a public property is called a *backing store* or *backing field*. You can declare `public` fields, but then you can't prevent code that uses your type from setting that field to an invalid value or otherwise changing an object's data.

Fields typically store the data that must be accessible to more than one type method and must be stored for longer than the lifetime of any single method. For example, a type that represents a calendar date might have three integer fields: one for the month, one for the day, and one for the year. Variables that aren't used outside the scope of a single method should be declared as *local variables* within the method body itself.

Fields are declared in the class or struct block by specifying the access level, followed by the type, followed by the name of the field. For example:

C#

```
public class CalendarEntry
{

    // private field (Located near wrapping "Date" property).
    private DateTime _date;

    // Public property exposes _date field safely.
    public DateTime Date
    {
        get
        {
```

```
            return _date;
        }
        set
        {
            // Set some reasonable boundaries for likely birth dates.
            if (value.Year > 1900 && value.Year <= DateTime.Today.Year)
            {
                _date = value;
            }
            else
            {
                throw new ArgumentOutOfRangeException("Date");
            }
        }
    }

    // public field (Generally not recommended).
    public string? Day;

    // Public method also exposes _date field safely.
    // Example call: birthday.SetDate("1975, 6, 30");
    public void SetDate(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        // Set some reasonable boundaries for likely birth dates.
        if (dt.Year > 1900 && dt.Year <= DateTime.Today.Year)
        {
            _date = dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }

    public TimeSpan GetTimeSpan(string dateString)
    {
        DateTime dt = Convert.ToDateTime(dateString);

        if (dt.Ticks < _date.Ticks)
        {
            return _date - dt;
        }
        else
        {
            throw new ArgumentOutOfRangeException("dateString");
        }
    }
}
```

To access a field in an instance, add a period after the instance name, followed by the name of the field, as in `instancename._fieldName`. For example:

C#

```
CalendarEntry birthday = new CalendarEntry();
birthday.Day = "Saturday";
```

A field can be given an initial value by using the assignment operator when the field is declared. To automatically assign the `Day` field to `"Monday"`, for example, you would declare `Day` as in the following example:

C#

```
public class CalendarDateWithInitialization
{
    public string Day = "Monday";
    //...
}
```

Fields are initialized immediately before the constructor for the object instance is called. If the constructor assigns the value of a field, it overwrites any value given during field declaration. For more information, see Using Constructors.

> **Note**
>
> A field initializer cannot refer to other instance fields.

Fields can be marked as public, private, protected, internal, protected internal, or private protected. These access modifiers define how users of the type can access the fields. For more information, see Access Modifiers.

A field can optionally be declared static. Static fields are available to callers at any time, even if no instance of the type exists. For more information, see Static Classes and Static Class Members.

A field can be declared readonly. A read-only field can only be assigned a value during initialization or in a constructor. A `static readonly` field is similar to a constant, except that the C# compiler doesn't have access to the value of a static read-only field at compile time, only at run time. For more information, see Constants.

A field can be declared required. A required field must be initialized by the constructor, or by an object initializers when an object is created. You add the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute attribute to any constructor declaration that initializes all required members.

The `required` modifier can't be combined with the `readonly` modifier on the same field. However, property can be `required` and `init` only.

Beginning with C# 12, Primary constructor parameters are an alternative to declaring fields. When your type has dependencies that must be provided at initialization, you can create a primary constructor that provides those dependencies. These parameters may be captured and used in place of declared fields in your types. In the case of record types, primary constructor parameters are surfaced as public properties.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- The C# type system
- Using Constructors
- Inheritance
- Access Modifiers
- Abstract and Sealed Classes and Class Members

Constants (C# Programming Guide)

Article • 03/12/2024

Constants are immutable values which are known at compile time and do not change for the life of the program. Constants are declared with the const modifier. Only the C# built-in types may be declared as `const`. Reference type constants other than String can only be initialized with a null value. User-defined types, including classes, structs, and arrays, cannot be `const`. Use the readonly modifier to create a class, struct, or array that is initialized one time at run time (for example in a constructor) and thereafter cannot be changed.

C# does not support `const` methods, properties, or events.

The enum type enables you to define named constants for integral built-in types (for example `int`, `uint`, `long`, and so on). For more information, see enum.

Constants must be initialized as they are declared. For example:

C#

```
class Calendar1
{
    public const int Months = 12;
}
```

In this example, the constant `Months` is always 12, and it cannot be changed even by the class itself. In fact, when the compiler encounters a constant identifier in C# source code (for example, `Months`), it substitutes the literal value directly into the intermediate language (IL) code that it produces. Because there is no variable address associated with a constant at run time, `const` fields cannot be passed by reference and cannot appear as an l-value in an expression.

> **Note**
>
> Use caution when you refer to constant values defined in other code such as DLLs. If a new version of the DLL defines a new value for the constant, your program will still hold the old literal value until it is recompiled against the new version.

Multiple constants of the same type can be declared at the same time, for example:

C#

```
class Calendar2
{
    public const int Months = 12, Weeks = 52, Days = 365;
}
```

The expression that is used to initialize a constant can refer to another constant if it does not create a circular reference. For example:

C#

```
class Calendar3
{
    public const int Months = 12;
    public const int Weeks = 52;
    public const int Days = 365;

    public const double DaysPerWeek = (double) Days / (double) Weeks;
    public const double DaysPerMonth = (double) Days / (double) Months;
}
```

Constants can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can access the constant. For more information, see Access Modifiers.

Constants are accessed as if they were static fields because the value of the constant is the same for all instances of the type. You do not use the `static` keyword to declare them. Expressions that are not in the class that defines the constant must use the class name, a period, and the name of the constant to access the constant. For example:

C#

```
int birthstones = Calendar.Months;
```

C# Language Specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Properties
- Types
- readonly

- Immutability in C# Part One: Kinds of Immutability

How to define abstract properties (C# Programming Guide)

Article • 03/12/2024

The following example shows how to define abstract properties. An abstract property declaration does not provide an implementation of the property accessors -- it declares that the class supports properties, but leaves the accessor implementation to derived classes. The following example demonstrates how to implement the abstract properties inherited from a base class.

This sample consists of three files, each of which is compiled individually and its resulting assembly is referenced by the next compilation:

- abstractshape.cs: the `Shape` class that contains an abstract `Area` property.
- shapes.cs: The subclasses of the `Shape` class.
- shapetest.cs: A test program to display the areas of some `Shape`-derived objects.

To compile the example, use the following command:

```
csc abstractshape.cs shapes.cs shapetest.cs
```

This will create the executable file shapetest.exe.

Examples

This file declares the `Shape` class that contains the `Area` property of the type `double`.

C#

```
// compile with: csc -target:library abstractshape.cs
public abstract class Shape
{
    public Shape(string s)
    {
        // calling the set accessor of the Id property.
        Id = s;
    }

    public string Id { get; set; }

    // Area is a read-only property - only a get accessor is needed:
    public abstract double Area
    {
```

```
        get;
    }

    public override string ToString()
    {
        return $"{Id} Area = {Area:F2}";
    }
}
```

- Modifiers on the property are placed on the property declaration itself. For example:

  ```
  C#

  public abstract double Area
  ```

- When declaring an abstract property (such as `Area` in this example), you simply indicate what property accessors are available, but do not implement them. In this example, only a get accessor is available, so the property is read-only.

The following code shows three subclasses of `Shape` and how they override the `Area` property to provide their own implementation.

```
C#

// compile with: csc -target:library -reference:abstractshape.dll shapes.cs
public class Square : Shape
{
    private int _side;

    public Square(int side, string id)
        : base(id)
    {
        _side = side;
    }

    public override double Area
    {
        get
        {
            // Given the side, return the area of a square:
            return _side * _side;
        }
    }
}

public class Circle : Shape
{
    private int _radius;
```

```
    public Circle(int radius, string id)
        : base(id)
    {
        _radius = radius;
    }

    public override double Area
    {
        get
        {
            // Given the radius, return the area of a circle:
            return _radius * _radius * Math.PI;
        }
    }
}

public class Rectangle : Shape
{
    private int _width;
    private int _height;

    public Rectangle(int width, int height, string id)
        : base(id)
    {
        _width = width;
        _height = height;
    }

    public override double Area
    {
        get
        {
            // Given the width and height, return the area of a rectangle:
            return _width * _height;
        }
    }
}
```

The following code shows a test program that creates a number of `Shape`-derived objects and prints out their areas.

C#

```
// compile with: csc -reference:abstractshape.dll;shapes.dll shapetest.cs
class TestClass
{
    static void Main()
    {
        Shape[] shapes =
        [
            new Square(5, "Square #1"),
            new Circle(3, "Circle #1"),
            new Rectangle( 4, 5, "Rectangle #1")
```

```
        ];

        Console.WriteLine("Shapes Collection");
        foreach (Shape s in shapes)
        {
            Console.WriteLine(s);
        }
    }
}
/* Output:
    Shapes Collection
    Square #1 Area = 25.00
    Circle #1 Area = 28.27
    Rectangle #1 Area = 20.00
*/
```

See also

- The C# type system
- Abstract and Sealed Classes and Class Members
- Properties

How to define constants in C#

Article • 10/27/2021

Constants are fields whose values are set at compile time and can never be changed. Use constants to provide meaningful names instead of numeric literals ("magic numbers") for special values.

> **Note**
>
> In C# the #define preprocessor directive cannot be used to define constants in the way that is typically used in C and C++.

To define constant values of integral types (`int`, `byte`, and so on) use an enumerated type. For more information, see enum.

To define non-integral constants, one approach is to group them in a single static class named `Constants`. This will require that all references to the constants be prefaced with the class name, as shown in the following example.

Example

C#

```
static class Constants
{
    public const double Pi = 3.14159;
    public const int SpeedOfLight = 300000; // km per sec.
}

class Program
{
    static void Main()
    {
        double radius = 5.3;
        double area = Constants.Pi * (radius * radius);
        int secsFromSun = 149476000 / Constants.SpeedOfLight; // in km
        Console.WriteLine(secsFromSun);
    }
}
```

The use of the class name qualifier helps ensure that you and others who use the constant understand that it is constant and cannot be modified.

See also

- The C# type system

Properties (C# Programming Guide)

A *property* is a member that provides a flexible mechanism to read, write, or compute the value of a data field. Properties appear as public data members, but they're implemented as special methods called *accessors*. This feature enables callers to access data easily and still helps promote data safety and flexibility. The syntax for properties is a natural extension to fields. A field defines a storage location:

C#

```
public class Person
{
    public string? FirstName;

    // Omitted for brevity.
}
```

Automatically implemented properties

A property definition contains declarations for a `get` and `set` accessor that retrieves and assigns the value of that property:

C#

```
public class Person
{
    public string? FirstName { get; set; }

    // Omitted for brevity.
}
```

The preceding example shows an *automatically implemented property*. The compiler generates a hidden backing field for the property. The compiler also implements the body of the `get` and `set` accessors. Any attributes are applied to the automatically implemented property. You can apply the attribute to the compiler-generated backing field by specifying the `field:` tag on the attribute.

You can initialize a property to a value other than the default by setting a value after the closing brace for the property. You might prefer the initial value for the `FirstName` property to be the empty string rather than `null`. You would specify that as shown in the following code:

C#

```
public class Person
{
```

```
    public string FirstName { get; set; } = string.Empty;

    // Omitted for brevity.
}
```

Field backed properties

In C# 14, you can add validation or other logic in the accessor for a property using the field keyword. The `field` keyword accesses the compiler synthesized backing field for a property. It enables you to write a property accessor without explicitly declaring a separate backing field.

C#

```
public class Person
{
    public string? FirstName
    {
        get;
        set => field = value.Trim();
    }

    // Omitted for brevity.
}
```

Required properties

The preceding example allows a caller to create a `Person` using the default constructor, without setting the `FirstName` property. The property changed type to a *nullable* string. You can *require* callers to set a property:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName) => FirstName = firstName;

    public required string FirstName { get; init; }

    // Omitted for brevity.
}
```

The preceding code makes two changes to the `Person` class. First, the `FirstName` property declaration includes the `required` modifier. That means any code that creates a new `Person`

must set this property using an object initializer. Second, the constructor that takes a `firstName` parameter has the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute attribute. This attribute informs the compiler that this constructor sets *all* `required` members. Callers using this constructor aren't required to set `required` properties with an object initializer.

> **ⓘ Important**
>
> Don't confuse `required` with *non-nullable*. It's valid to set a `required` property to `null` or `default`. If the type is non-nullable, such as `string` in these examples, the compiler issues a warning.

C#

```
var aPerson = new Person("John");
aPerson = new Person { FirstName = "John"};
// Error CS9035: Required member `Person.FirstName` must be set:
//aPerson2 = new Person();
```

Expression body definitions

Property accessors often consist of single-line statements. The accessors assign or return the result of an expression. You can implement these properties as expression-bodied members. Expression body definitions consist of the `=>` token followed by the expression to assign to or retrieve from the property.

Read-only properties can implement the `get` accessor as an expression-bodied member. The following example implements the read-only `Name` property as an expression-bodied member:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
    }

    public required string FirstName { get; init; }
    public required string LastName { get; init; }
```

```
    public string Name => $"{FirstName} {LastName}";

    // Omitted for brevity.
}
```

The `Name` property is a computed property. There's no backing field for `Name`. The property computes it each time.

Access control

The preceding examples showed read / write properties. You can also create read-only properties, or give different accessibility to the set and get accessors. Suppose that your `Person` class should only enable changing the value of the `FirstName` property from other methods in the class. You could give the set accessor `private` accessibility instead of `internal` or `public`:

C#

```
public class Person
{
    public string? FirstName { get; private set; }

    // Omitted for brevity.
}
```

The `FirstName` property can be read from any code, but it can be assigned only from code in the `Person` class.

You can add any restrictive access modifier to either the set or get accessors. An access modifier on an individual accessor must be more restrictive than the access of the property. The preceding code is legal because the `FirstName` property is `public`, but the set accessor is `private`. You couldn't declare a `private` property with a `public` accessor. Property declarations can also be declared `protected`, `internal`, `protected internal`, or, even `private`.

There are two special access modifiers for `set` accessors:

- A `set` accessor can have `init` as its access modifier. That `set` accessor can be called only from an object initializer or the type's constructors. It's more restrictive than `private` on the `set` accessor.
- An automatically implemented property can declare a `get` accessor without a `set` accessor. In that case, the compiler allows the `set` accessor to be called only from the type's constructors. It's more restrictive than the `init` accessor on the `set` accessor.

Modify the `Person` class so as follows:

C#

```
public class Person
{
    public Person(string firstName) => FirstName = firstName;

    public string FirstName { get; }

    // Omitted for brevity.
}
```

The preceding example requires callers to use the constructor that includes the `FirstName` parameter. Callers can't use object initializers to assign a value to the property. To support initializers, you can make the `set` accessor an `init` accessor, as shown in the following code:

C#

```
public class Person
{
    public Person() { }
    public Person(string firstName) => FirstName = firstName;

    public string? FirstName { get; init; }

    // Omitted for brevity.
}
```

These modifiers are often used with the `required` modifier to force proper initialization.

Properties with backing fields

You can mix the concept of a computed property with a private field and create a *cached evaluated property*. For example, update the `FullName` property so that the string formatting happens on the first access:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName)
    {
        FirstName = firstName;
        LastName = lastName;
```

```
    }

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    private string? _fullName;
    public string FullName
    {
        get
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

This implementation works because the `FirstName` and `LastName` properties are readonly. People can change their name. Updating the `FirstName` and `LastName` properties to allow `set` accessors requires you to invalidate any cached value for `fullName`. You modify the `set` accessors of the `FirstName` and `LastName` property so the `fullName` field is calculated again:

C#

```
public class Person
{
    private string? _firstName;
    public string? FirstName
    {
        get => _firstName;
        set
        {
            _firstName = value;
            _fullName = null;
        }
    }

    private string? _lastName;
    public string? LastName
    {
        get => _lastName;
        set
        {
            _lastName = value;
            _fullName = null;
        }
    }

    private string? _fullName;
    public string FullName
    {
        get
```

```
        {
            if (_fullName is null)
                _fullName = $"{FirstName} {LastName}";
            return _fullName;
        }
    }
}
```

This final version evaluates the `FullName` property only when needed. The previously calculated version is used if valid. Otherwise, the calculation updates the cached value. Developers using this class don't need to know the details of the implementation. None of these internal changes affect the use of the Person object.

Beginning with C# 13, you can create partial properties in `partial` classes. The implementing declaration for a `partial` property can't be an automatically implemented property. An automatically implemented property uses the same syntax as a declaring partial property declaration.

Properties

Properties are a form of smart fields in a class or object. From outside the object, they appear like fields in the object. However, properties can be implemented using the full palette of C# functionality. You can provide validation, different accessibility, lazy evaluation, or any requirements your scenarios need.

- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as automatically implemented properties.
- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. An init property accessor is used to assign a new value only during object construction. These accessors can have different access levels. For more information, see Restricting Accessor Accessibility.
- The value keyword is used to define the value the `set` or `init` accessor is assigning.
- Properties can be *read-write* (they have both a `get` and a `set` accessor), *read-only* (they have a `get` accessor but no `set` accessor), or *write-only* (they have a `set` accessor, but no `get` accessor). Write-only properties are rare.

C# Language Specification

For more information, see Properties in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- Indexers
- init keyword
- get keyword
- set keyword

Last updated on 11/19/2025

Using Properties (C# Programming Guide)

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field; accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a get accessor and/or a set or init accessor. The code block for the `get` accessor is executed when the property is read; the code block for the `set` or `init` accessor is executed when the property is assigned a value. A property without a `set` accessor is considered read-only. A property without a `get` accessor is considered write-only. A property that has both accessors is read-write. You can use an `init` accessor instead of a `set` accessor to enable the property to be set as part of object initialization but otherwise make it read-only.

Unlike fields, properties aren't classified as variables. Therefore, you can't pass a property as a ref or out parameter.

Properties have many uses:

- They can validate data before allowing a change.
- They can transparently expose data on a class where that data is retrieved from some other source, such as a database.
- They can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a `get`-accessor and/or a `set` accessor. For example:

C#

```
public class Date
{
    private int _month = 7;  // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

In this example, `Month` is declared as a property so that the `set` accessor can make sure that the `Month` value is set between 1 and 12. The `Month` property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It's common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property. For more information about public and private access restrictions, see Access Modifiers. Automatically implemented properties provide simplified syntax for simple property declarations. For more information, see Automatically implemented properties.

Beginning with C# 14, you can use field backed properties to add validation to the `set` accessor of an automatically implemented property, as shown in the following example:

C#

```
public class DateExample
{
    public int Month
    {
        get;
        set
        {
            if ((value > 0) && (value < 13))
            {
                field = value;
            }
        }
    }
}
```

The get accessor

The body of the `get` accessor resembles that of a method. It must return a value of the property type. The C# compiler and Just-in-time (JIT) compiler detect common patterns for implementing the `get` accessor, and optimizes those patterns. For example, a `get` accessor that returns a field without performing any computation is likely optimized to a memory read of that field. Automatically implemented properties follow this pattern and benefit from these optimizations. However, a virtual `get` accessor method can't be inlined because the compiler doesn't know at compile time which method might actually be called at run time. The following example shows a `get` accessor that returns the value of a private field `_name`:

C#

```
class Employee
{
    private string _name;           // the name field
```

```
    public string Name => _name;    // the Name property
}
```

When you reference the property, except as the target of an assignment, the `get` accessor is invoked to read the value of the property. For example:

C#

```
var employee = new Employee();
//...

System.Console.Write(employee.Name); // the get accessor is invoked here
```

The `get` accessor must be an expression-bodied member, or end in a return or throw statement, and control can't flow off the accessor body.

> ⚠ **Warning**
>
> It's generally a bad programming style to change the state of the object by using the `get` accessor. One exception to this rule is a *lazy evaluated* property, where the value of a property is computed only when it's first accessed.

The `get` accessor can be used to return the field value or to compute it and return it. For example:

C#

```
class Manager
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

In the previous example, if you don't assign a value to the `Name` property, it returns the value `NA`.

The set accessor

The `set` accessor resembles a method whose return type is void. It uses an implicit parameter called `value`, whose type is the type of the property. The compiler and JIT compiler also recognize common patterns for a `set` or `init` accessor. Those common patterns are optimized, directly writing the memory for the backing field. In the following example, a `set` accessor is added to the `Name` property:

C#

```
class Student
{
    private string _name;       // the name field
    public string Name          // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

When you assign a value to the property, the `set` accessor is invoked by using an argument that provides the new value. For example:

C#

```
var student = new Student();
student.Name = "Joe"; // the set accessor is invoked here

System.Console.Write(student.Name); // the get accessor is invoked here
```

It's an error to use the implicit parameter name, `value`, for a local variable declaration in a `set` accessor.

The init accessor

The code to create an `init` accessor is the same as the code to create a `set` accessor except that you use the `init` keyword instead of `set`. The difference is that the `init` accessor can only be used in the constructor or by using an object-initializer.

Remarks

Properties can be marked as `public`, `private`, `protected`, `internal`, `protected internal`, or `private protected`. These access modifiers define how users of the class can access the property. The `get` and `set` accessors for the same property can have different access modifiers. For example, the `get` might be `public` to allow read-only access from outside the type, and the `set` can be `private` or `protected`. For more information, see Access Modifiers.

A property can be declared as a static property by using the `static` keyword. Static properties are available to callers at any time, even if no instance of the class exists. For more information, see Static Classes and Static Class Members.

A property can be marked as a virtual property by using the virtual keyword. Virtual properties enable derived classes to override the property behavior by using the override keyword. For more information about these options, see Inheritance.

A property overriding a virtual property can also be sealed, specifying that for derived classes it's no longer virtual. Lastly, a property can be declared abstract. Abstract properties don't define an implementation in the class, and derived classes must write their own implementation. For more information about these options, see Abstract and Sealed Classes and Class Members.

> **Note**
>
> It's an error to use a **virtual**, **abstract**, or **override** modifier on an accessor of a **static** property.

Examples

This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments `NumberOfEmployees` by 1, and displays the Employee name and number.

C#

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's
number:
}
```

Hidden property example

This example demonstrates how to access a property in a base class that's hidden by another property that has the same name in a derived class:

C#

```
public class Employee
{
    private string _name;
    public string Name
    {
        get => _name;
        set => _name = value;
    }
}

public class Manager : Employee
{
    private string _name;

    // Notice the use of the new modifier:
    public new string Name
    {
        get => _name;
        set => _name = value + ", Manager";
    }
}

class TestHiding
{
    public static void Test()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine($"Name in the derived class is: {m1.Name}");
        System.Console.WriteLine($"Name in the base class is:
{((Employee)m1).Name}");
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

The following are important points in the previous example:

- The property `Name` in the derived class hides the property `Name` in the base class. In such a case, the `new` modifier is used in the declaration of the property in the derived class:

  C#

  ```
  public new string Name
  ```

- The cast `(Employee)` is used to access the hidden property in the base class:

  C#

  ```
  ((Employee)m1).Name = "Mary";
  ```

For more information about hiding members, see the new Modifier.

Override property example

In this example, two classes, `Cube` and `Square`, implement an abstract class, `Shape`, and override its abstract `Area` property. Note the use of the override modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the corresponding side for the square and cube.

C#

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}
```

```
class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine($"Area of the square = {s.Area:F2}");
        System.Console.WriteLine($"Area of the cube = {c.Area:F2}");
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
        c.Area = area;

        // Display the results:
        System.Console.WriteLine($"Side of the square = {s.side:F2}");
        System.Console.WriteLine($"Side of the cube = {c.side:F2}");
    }
}
/* Example Output:
    Enter the side: 4
    Area of the square = 16.00
    Area of the cube = 96.00

    Enter the area: 24
    Side of the square = 4.90
    Side of the cube = 2.00
*/
```

See also

- Properties
- Interface properties
- Automatically implemented properties
- Partial properties

Last updated on 11/19/2025

Interface Properties (C# Programming Guide)

Article • 10/26/2024

Properties can be declared on an interface. The following example declares an interface property accessor:

C#

```
public interface ISampleInterface
{
    // Property declaration:
    string Name
    {
        get;
        set;
    }
}
```

Interface properties typically don't have a body. The accessors indicate whether the property is read-write, read-only, or write-only. Unlike in classes and structs, declaring the accessors without a body doesn't declare an automatically implemented property. An interface can define a default implementation for members, including properties. Defining a default implementation for a property in an interface is rare because interfaces can't define instance data fields.

Example

In this example, the interface `IEmployee` has a read-write property, `Name`, and a read-only property, `Counter`. The class `Employee` implements the `IEmployee` interface and uses these two properties. The program reads the name of a new employee and the current number of employees and displays the employee name and the computed employee number.

You could use the fully qualified name of the property, which references the interface in which the member is declared. For example:

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
```

```
    set { }
}
```

The preceding example demonstrates Explicit Interface Implementation. For example, if the class `Employee` is implementing two interfaces `ICitizen` and `IEmployee` and both interfaces have the `Name` property, the explicit interface member implementation is necessary. That is, the following property declaration:

C#

```
string IEmployee.Name
{
    get { return "Employee Name"; }
    set { }
}
```

Implements the `Name` property on the `IEmployee` interface, while the following declaration:

C#

```
string ICitizen.Name
{
    get { return "Citizen Name"; }
    set { }
}
```

Implements the `Name` property on the `ICitizen` interface.

C#

```
interface IEmployee
{
    string Name
    {
        get;
        set;
    }

    int Counter
    {
        get;
    }
}

public class Employee : IEmployee
{
    public static int numberOfEmployees;
```

```
    private string _name;
    public string Name  // read-write instance property
    {
        get => _name;
        set => _name = value;
    }

    private int _counter;
    public int Counter  // read-only instance property
    {
        get => _counter;
    }

    // constructor
    public Employee() => _counter = ++numberOfEmployees;
}
```

C#

```
System.Console.Write("Enter number of employees: ");
Employee.numberOfEmployees = int.Parse(System.Console.ReadLine());

Employee e1 = new Employee();
System.Console.Write("Enter the name of the new employee: ");
e1.Name = System.Console.ReadLine();

System.Console.WriteLine("The employee information:");
System.Console.WriteLine($"Employee number: {e1.Counter}");
System.Console.WriteLine($"Employee name: {e1.Name}");
```

Sample output

Console

```
Enter number of employees: 210
Enter the name of the new employee: Hazem Abolrous
The employee information:
Employee number: 211
Employee name: Hazem Abolrous
```

See also

- Properties
- Using Properties
- Comparison Between Properties and Indexers
- Indexers

- Interfaces

Restricting Accessor Accessibility (C# Programming Guide)

Article • 10/30/2024

The get and set portions of a property or indexer are called *accessors*. By default these accessors have the same visibility or access level of the property or indexer to which they belong. For more information, see accessibility levels. However, it's sometimes useful to restrict access to one of these accessors. Typically, you restrict the accessibility of the `set` accessor, while keeping the `get` accessor publicly accessible. For example:

C#

```
private string _name = "Hello";

public string Name
{
    get
    {
        return _name;
    }
    protected set
    {
        _name = value;
    }
}
```

In this example, a property called `Name` defines a `get` and `set` accessor. The `get` accessor receives the accessibility level of the property itself, `public` in this case, while the `set` accessor is explicitly restricted by applying the protected access modifier to the accessor itself.

> **Note**
>
> The examples in this article don't use automatically implemented properties. *Automatically implemented properties* provide a concise syntax for declaring properties when a custom backing field isn't required.

Restrictions on Access Modifiers on Accessors

Using the accessor modifiers on properties or indexers is subject to these conditions:

- You can't use accessor modifiers on an interface or an explicit interface member implementation.
- You can use accessor modifiers only if the property or indexer has both `set` and `get` accessors. In this case, the modifier is permitted on only one of the two accessors.
- If the property or indexer has an override modifier, the accessor modifier must match the accessor of the overridden accessor, if any.
- The accessibility level on the accessor must be more restrictive than the accessibility level on the property or indexer itself.

Access Modifiers on Overriding Accessors

When you override a property or indexer, the overridden accessors must be accessible to the overriding code. Also, the accessibility of both the property/indexer and its accessors must match the corresponding overridden property/indexer and its accessors. For example:

C#

```
public class Parent
{
    public virtual int TestProperty
    {
        // Notice the accessor accessibility level.
        protected set { }

        // No access modifier is used here.
        get { return 0; }
    }
}
public class Kid : Parent
{
    public override int TestProperty
    {
        // Use the same accessibility level as in the overridden accessor.
        protected set { }

        // Cannot use access modifier here.
        get { return 0; }
    }
}
```

Implementing Interfaces

When you use an accessor to implement an interface, the accessor may not have an access modifier. However, if you implement the interface using one accessor, such as `get`, the other accessor can have an access modifier, as in the following example:

C#

```
public interface ISomeInterface
{
    int TestProperty
    {
        // No access modifier allowed here
        // because this is an interface.
        get;
    }
}

public class TestClass : ISomeInterface
{
    public int TestProperty
    {
        // Cannot use access modifier here because
        // this is an interface implementation.
        get { return 10; }

        // Interface property does not have set accessor,
        // so access modifier is allowed.
        protected set { }
    }
}
```

Accessor Accessibility Domain

If you use an access modifier on the accessor, the accessibility domain of the accessor is determined by this modifier.

If you didn't use an access modifier on the accessor, the accessibility domain of the accessor is determined by the accessibility level of the property or indexer.

Example

The following example contains three classes, `BaseClass`, `DerivedClass`, and `MainClass`. There are two properties on the `BaseClass`, `Name` and `Id` on both classes. The example demonstrates how the property `Id` on `DerivedClass` can be hidden by the property `Id` on `BaseClass` when you use a restrictive access modifier such as protected or private. Therefore, when you assign values to this property, the property on the `BaseClass` class

is called instead. Replacing the access modifier by public will make the property accessible.

The example also demonstrates that a restrictive access modifier, such as `private` or `protected`, on the `set` accessor of the `Name` property in `DerivedClass` prevents access to the accessor in the derived class. It generates an error when you assign to it, or accesses the base class property of the same name, if it's accessible.

C#

```
public class BaseClass
{
    private string _name = "Name-BaseClass";
    private string _id = "ID-BaseClass";

    public string Name
    {
        get { return _name; }
        set { }
    }

    public string Id
    {
        get { return _id; }
        set { }
    }
}

public class DerivedClass : BaseClass
{
    private string _name = "Name-DerivedClass";
    private string _id = "ID-DerivedClass";

    new public string Name
    {
        get
        {
            return _name;
        }

        // Using "protected" would make the set accessor not accessible.
        set
        {
            _name = value;
        }
    }

    // Using private on the following property hides it in the Main Class.
    // Any assignment to the property will use Id in BaseClass.
    new private string Id
    {
        get
```

```
        {
            return _id;
        }
        set
        {
            _id = value;
        }
    }
}

class MainClass
{
    static void Main()
    {
        BaseClass b1 = new BaseClass();
        DerivedClass d1 = new DerivedClass();

        b1.Name = "Mary";
        d1.Name = "John";

        b1.Id = "Mary123";
        d1.Id = "John123";  // The BaseClass.Id property is called.

        System.Console.WriteLine("Base: {0}, {1}", b1.Name, b1.Id);
        System.Console.WriteLine("Derived: {0}, {1}", d1.Name, d1.Id);

        // Keep the console window open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
/* Output:
    Base: Name-BaseClass, ID-BaseClass
    Derived: John, ID-BaseClass
*/
```

Comments

Notice that if you replace the declaration `new private string Id` by `new public string Id`, you get the output:

`Name and ID in the base class: Name-BaseClass, ID-BaseClass` `Name and ID in the derived class: John, John123`

See also

- Properties
- Indexers

- Access Modifiers
- Init only properties
- Required properties

How to declare and use read write properties (C# Programming Guide)

Article • 07/30/2022

Properties provide the convenience of public data members without the risks that come with unprotected, uncontrolled, and unverified access to an object's data. Properties declare *accessors*: special methods that assign and retrieve values from the underlying data member. The set accessor enables data members to be assigned, and the get accessor retrieves data member values.

This sample shows a `Person` class that has two properties: `Name` (string) and `Age` (int). Both properties provide `get` and `set` accessors, so they're considered read/write properties.

Example

C#

```
class Person
{
    private string _name = "N/A";
    private int _age = 0;

    // Declare a Name property of type string:
    public string Name
    {
        get
        {
            return _name;
        }
        set
        {
            _name = value;
        }
    }

    // Declare an Age property of type int:
    public int Age
    {
        get
        {
            return _age;
        }

        set
        {
```

```
            _age = value;
        }
    }

    public override string ToString()
    {
        return "Name = " + Name + ", Age = " + Age;
    }
}

public class Wrapper
{
    private string _name = "N/A";
    public string Name
    {
        get
        {
            return _name;
        }
        private set
        {
            _name = value;
        }
    }

}

class TestPerson
{
    static void Main()
    {
        // Create a new Person object:
        Person person = new Person();

        // Print out the name and the age associated with the person:
        Console.WriteLine($"Person details - {person}");

        // Set some values on the person object:
        person.Name = "Joe";
        person.Age = 99;
        Console.WriteLine($"Person details - {person}");

        // Increment the Age property:
        person.Age += 1;
        Console.WriteLine($"Person details - {person}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
    Person details - Name = N/A, Age = 0
    Person details - Name = Joe, Age = 99
```

```
    Person details - Name = Joe, Age = 100
*/
```

Robust Programming

In the previous example, the `Name` and `Age` properties are public and include both a `get` and a `set` accessor. Public accessors allow any object to read and write these properties. It's sometimes desirable, however, to exclude one of the accessors. You can omit the `set` accessor to make the property read-only:

C#

```
public string Name
{
    get
    {
        return _name;
    }
    private set
    {
        _name = value;
    }
}
```

Alternatively, you can expose one accessor publicly but make the other private or protected. For more information, see Asymmetric Accessor Accessibility.

Once the properties are declared, they can be used as fields of the class. Properties allow for a natural syntax when both getting and setting the value of a property, as in the following statements:

C#

```
person.Name = "Joe";
person.Age = 99;
```

In a property `set` method a special `value` variable is available. This variable contains the value that the user specified, for example:

C#

```
_name = value;
```

Notice the clean syntax for incrementing the `Age` property on a `Person` object:

C#

```
person.Age += 1;
```

If separate `set` and `get` methods were used to model properties, the equivalent code might look like this:

C#

```
person.SetAge(person.GetAge() + 1);
```

The `ToString` method is overridden in this example:

C#

```
public override string ToString()
{
    return "Name = " + Name + ", Age = " + Age;
}
```

Notice that `ToString` isn't explicitly used in the program. It's invoked by default by the `WriteLine` calls.

See also

- Properties
- The C# type system

Automatically implemented properties

Automatically implemented properties make property-declaration more concise when no other logic is required in the property accessors. They also enable client code to create objects. When you declare a property as shown in the following example, the compiler creates a private, anonymous backing field that can only be accessed through the property's `get` and `set` accessors. `init` accessors can also be declared as automatically implemented properties.

The following example shows a simple class that has some automatically implemented properties:

C#

```
// This class is mutable. Its data can be modified from
// outside the class.
public class Customer
{
    // Auto-implemented properties for trivial get and set
    public double TotalPurchases { get; set; }
    public string Name { get; set; }
    public int CustomerId { get; set; }

    // Constructor
    public Customer(double purchases, string name, int id)
    {
        TotalPurchases = purchases;
        Name = name;
        CustomerId = id;
    }

    // Methods
    public string GetContactInfo() { return "ContactInfo"; }
    public string GetTransactionHistory() { return "History"; }

    // .. Additional methods, events, etc.
}

class Program
{
    static void Main()
    {
        // Initialize a new object.
        Customer cust1 = new Customer(4987.63, "Northwind", 90108);

        // Modify a property.
        cust1.TotalPurchases += 499.99;
    }
}
```

You can't declare automatically implemented properties in interfaces. Automatically implemented and field backed properties declare a private instance backing field, and interfaces can't declare instance fields. Declaring a property in an interface without defining a body declares a property with accessors. Each type that implements that interface must implement that property.

You can initialize automatically implemented properties similarly to fields:

C#

```
public string FirstName { get; set; } = "Jane";
```

The class that is shown in the previous example is mutable. Client code can change the values in objects after creation. In complex classes that contain significant behavior (methods) and data, it's often necessary to have public properties. However, for small classes or structs that just encapsulate a set of values (data) and have little or no behaviors, you should use one of the following options for making the objects immutable:

- Declare only a `get` accessor (immutable everywhere except the constructor).
- Declare a `get` accessor and an `init` accessor (immutable everywhere except during object construction).
- Declare the `set` accessor as private (immutable to consumers).

For more information, see How to implement a lightweight class with automatically implemented properties.

You might need to add validation to an automatically implemented property. C# 14 adds field backed properties. You use the `field` keyword to access the compiler synthesized backing field of an automatically implemented property. For example, you could ensure that the `FirstName` property in the preceding example can't be set to `null` or the empty string:

C#

```
public string FirstName
{
    get;
    set
    {
        field = (string.IsNullOrWhiteSpace(value) is false
            ? value
            : throw new ArgumentException(nameof(value), "First name can't be
whitespace or null"));
    }
} = "Jane";
```

This feature enables you to add logic to accessors without requiring you to explicitly declare the backing field. You use the `field` keyword to access the backing field generated by the compiler.

See also

- Use auto-implemented properties (style rule IDE0032)
- Properties
- Modifiers

Last updated on 11/19/2025

How to implement a lightweight class with automatically implemented properties

Article • 10/26/2024

This example shows how to create an immutable lightweight class that serves only to encapsulate a set of automatically implemented properties. Use this kind of construct instead of a struct when you must use reference type semantics.

You can make an immutable property in the following ways:

- Declare only the get accessor, which makes the property immutable everywhere except in the type's constructor.
- Declare an init accessor instead of a `set` accessor, which makes the property settable only in the constructor or by using an object initializer.
- Declare the set accessor to be private. The property is settable within the type, but it's immutable to consumers.

You can add the required modifier to the property declaration to force callers to set the property as part of initializing a new object.

The following example shows how a property with only get accessor differs than one with get and private set.

C#

```
class Contact
{
    public string Name { get; }
    public string Address { get; private set; }

    public Contact(string contactName, string contactAddress)
    {
        // Both properties are accessible in the constructor.
        Name = contactName;
        Address = contactAddress;
    }

    // Name isn't assignable here. This will generate a compile error.
    //public void ChangeName(string newName) => Name = newName;

    // Address is assignable here.
    public void ChangeAddress(string newAddress) => Address = newAddress;
}
```

Example

The following example shows two ways to implement an immutable class that has automatically implemented properties. Each way declares one of the properties with a private `set` and one of the properties with a `get` only. The first class uses a constructor only to initialize the properties, and the second class uses a static factory method that calls a constructor.

C#

```
// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// constructor to initialize its properties.
class Contact
{
    // Read-only property.
    public string Name { get; }

    // Read-write property with a private set accessor.
    public string Address { get; private set; }

    // Public constructor.
    public Contact(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }
}

// This class is immutable. After an object is created,
// it cannot be modified from outside the class. It uses a
// static method and private constructor to initialize its properties.
public class Contact2
{
    // Read-write property with a private set accessor.
    public string Name { get; private set; }

    // Read-only property.
    public string Address { get; }

    // Private constructor.
    private Contact2(string contactName, string contactAddress)
    {
        Name = contactName;
        Address = contactAddress;
    }

    // Public factory method.
    public static Contact2 CreateContact(string name, string address)
    {
        return new Contact2(name, address);
    }
```

```
}

public class Program
{
    static void Main()
    {
        // Some simple data sources.
        string[] names = ["Terry Adams","Fadi Fakhouri", "Hanying Feng",
                          "Cesar Garcia", "Debra Garcia"];
        string[] addresses = ["123 Main St.", "345 Cypress Ave.", "678 1st
Ave",
                              "12 108th St.", "89 E. 42nd St."];

        // Simple query to demonstrate object creation in select clause.
        // Create Contact objects by using a constructor.
        var query1 = from i in Enumerable.Range(0, 5)
                     select new Contact(names[i], addresses[i]);

        // List elements cannot be modified by client code.
        var list = query1.ToList();
        foreach (var contact in list)
        {
            Console.WriteLine("{0}, {1}", contact.Name, contact.Address);
        }

        // Create Contact2 objects by using a static factory method.
        var query2 = from i in Enumerable.Range(0, 5)
                     select Contact2.CreateContact(names[i],
addresses[i]);

        // Console output is identical to query1.
        var list2 = query2.ToList();

        // List elements cannot be modified by client code.
        // CS0272:
        // list2[0].Name = "Eugene Zabokritski";
    }
}

/* Output:
    Terry Adams, 123 Main St.
    Fadi Fakhouri, 345 Cypress Ave.
    Hanying Feng, 678 1st Ave
    Cesar Garcia, 12 108th St.
    Debra Garcia, 89 E. 42nd St.
*/
```

The compiler creates backing fields for each automatically implemented property. The fields aren't accessible directly from source code.

See also

- Properties
- struct
- Object and Collection Initializers

Methods (C# Programming Guide)

Article • 06/20/2023

A method is a code block that contains a series of statements. A program causes the statements to be executed by calling the method and specifying any required method arguments. In C#, every executed instruction is performed in the context of a method.

The `Main` method is the entry point for every C# application and it's called by the common language runtime (CLR) when the program is started. In an application that uses top-level statements, the `Main` method is generated by the compiler and contains all top-level statements.

> **Note**
>
> This article discusses named methods. For information about anonymous functions, see Lambda expressions.

Method signatures

Methods are declared in a class, struct, or interface by specifying the access level such as `public` or `private`, optional modifiers such as `abstract` or `sealed`, the return value, the name of the method, and any method parameters. These parts together are the signature of the method.

> **Important**
>
> A return type of a method is not part of the signature of the method for the purposes of method overloading. However, it is part of the signature of the method when determining the compatibility between a delegate and the method that it points to.

Method parameters are enclosed in parentheses and are separated by commas. Empty parentheses indicate that the method requires no parameters. This class contains four methods:

C#

```
abstract class Motorcycle
{
    // Anyone can call this.
```

```
    public void StartEngine() {/* Method statements here */ }

    // Only derived classes can call this.
    protected void AddGas(int gallons) { /* Method statements here */ }

    // Derived classes can override the base class implementation.
    public virtual int Drive(int miles, int speed) { /* Method statements
here */ return 1; }

    // Derived classes must implement this.
    public abstract double GetTopSpeed();
}
```

Method access

Calling a method on an object is like accessing a field. After the object name, add a period, the name of the method, and parentheses. Arguments are listed within the parentheses, and are separated by commas. The methods of the `Motorcycle` class can therefore be called as in the following example:

C#

```
class TestMotorcycle : Motorcycle
{
    public override double GetTopSpeed()
    {
        return 108.4;
    }

    static void Main()
    {
        TestMotorcycle moto = new TestMotorcycle();

        moto.StartEngine();
        moto.AddGas(15);
        moto.Drive(5, 20);
        double speed = moto.GetTopSpeed();
        Console.WriteLine($"My top speed is {speed}");
    }
}
```

Method parameters vs. arguments

The method definition specifies the names and types of any parameters that are required. When calling code calls the method, it provides concrete values called arguments for each parameter. The arguments must be compatible with the parameter

type but the argument name (if any) used in the calling code doesn't have to be the same as the parameter named defined in the method. For example:

C#

```
public void Caller()
{
    int numA = 4;
    // Call with an int variable.
    int productA = Square(numA);

    int numB = 32;
    // Call with another int variable.
    int productB = Square(numB);

    // Call with an integer literal.
    int productC = Square(12);

    // Call with an expression that evaluates to int.
    productC = Square(productA * 3);
}

int Square(int i)
{
    // Store input argument in a local variable.
    int input = i;
    return input * input;
}
```

Passing by reference vs. passing by value

By default, when an instance of a value type is passed to a method, its copy is passed instead of the instance itself. Therefore, changes to the argument have no effect on the original instance in the calling method. To pass a value-type instance by reference, use the `ref` keyword. For more information, see Passing Value-Type Parameters.

When an object of a reference type is passed to a method, a reference to the object is passed. That is, the method receives not the object itself but an argument that indicates the location of the object. If you change a member of the object by using this reference, the change is reflected in the argument in the calling method, even if you pass the object by value.

You create a reference type by using the `class` keyword, as the following example shows:

C#

```
public class SampleRefType
{
    public int value;
}
```

Now, if you pass an object that is based on this type to a method, a reference to the object is passed. The following example passes an object of type `SampleRefType` to method `ModifyObject`:

C#

```
public static void TestRefType()
{
    SampleRefType rt = new SampleRefType();
    rt.value = 44;
    ModifyObject(rt);
    Console.WriteLine(rt.value);
}

static void ModifyObject(SampleRefType obj)
{
    obj.value = 33;
}
```

The example does essentially the same thing as the previous example in that it passes an argument by value to a method. But, because a reference type is used, the result is different. The modification that is made in `ModifyObject` to the `value` field of the parameter, `obj`, also changes the `value` field of the argument, `rt`, in the `TestRefType` method. The `TestRefType` method displays 33 as the output.

For more information about how to pass reference types by reference and by value, see Passing Reference-Type Parameters and Reference Types.

Return values

Methods can return a value to the caller. If the return type (the type listed before the method name) is not `void`, the method can return the value by using the return statement. A statement with the `return` keyword followed by a value that matches the return type will return that value to the method caller.

The value can be returned to the caller by value or by reference. Values are returned to the caller by reference if the `ref` keyword is used in the method signature and it follows each `return` keyword. For example, the following method signature and return

statement indicate that the method returns a variable named `estDistance` by reference to the caller.

C#

```
public ref double GetEstimatedDistance()
{
    return ref estDistance;
}
```

The `return` keyword also stops the execution of the method. If the return type is `void`, a `return` statement without a value is still useful to stop the execution of the method. Without the `return` keyword, the method will stop executing when it reaches the end of the code block. Methods with a non-void return type are required to use the `return` keyword to return a value. For example, these two methods use the `return` keyword to return integers:

C#

```
class SimpleMath
{
    public int AddTwoNumbers(int number1, int number2)
    {
        return number1 + number2;
    }

    public int SquareANumber(int number)
    {
        return number * number;
    }
}
```

To use a value returned from a method, the calling method can use the method call itself anywhere a value of the same type would be sufficient. You can also assign the return value to a variable. For example, the following two code examples accomplish the same goal:

C#

```
int result = obj.AddTwoNumbers(1, 2);
result = obj.SquareANumber(result);
// The result is 9.
Console.WriteLine(result);
```

C#

```
result = obj.SquareANumber(obj.AddTwoNumbers(1, 2));
// The result is 9.
Console.WriteLine(result);
```

Using a local variable, in this case, `result`, to store a value is optional. It may help the readability of the code, or it may be necessary if you need to store the original value of the argument for the entire scope of the method.

To use a value returned by reference from a method, you must declare a ref local variable if you intend to modify its value. For example, if the `Planet.GetEstimatedDistance` method returns a Double value by reference, you can define it as a ref local variable with code like the following:

C#

```
ref double distance = ref Planet.GetEstimatedDistance();
```

Returning a multi-dimensional array from a method, `M`, that modifies the array's contents is not necessary if the calling function passed the array into `M`. You may return the resulting array from `M` for good style or functional flow of values, but it is not necessary because C# passes all reference types by value, and the value of an array reference is the pointer to the array. In the method `M`, any changes to the array's contents are observable by any code that has a reference to the array, as shown in the following example:

C#

```
static void Main(string[] args)
{
    int[,] matrix = new int[2, 2];
    FillMatrix(matrix);
    // matrix is now full of -1
}

public static void FillMatrix(int[,] matrix)
{
    for (int i = 0; i < matrix.GetLength(0); i++)
    {
        for (int j = 0; j < matrix.GetLength(1); j++)
        {
            matrix[i, j] = -1;
        }
    }
}
```

Async methods

By using the async feature, you can invoke asynchronous methods without using explicit callbacks or manually splitting your code across multiple methods or lambda expressions.

If you mark a method with the async modifier, you can use the await operator in the method. When control reaches an await expression in the async method, control returns to the caller, and progress in the method is suspended until the awaited task completes. When the task is complete, execution can resume in the method.

> **Note**
>
> An async method returns to the caller when either it encounters the first awaited object that's not yet complete or it gets to the end of the async method, whichever occurs first.

An async method typically has a return type of Task<TResult>, Task, IAsyncEnumerable<T> or `void`. The `void` return type is used primarily to define event handlers, where a `void` return type is required. An async method that returns `void` can't be awaited, and the caller of a void-returning method can't catch exceptions that the method throws. An async method can have any task-like return type.

In the following example, `DelayAsync` is an async method that has a return type of Task<TResult>. `DelayAsync` has a `return` statement that returns an integer. Therefore the method declaration of `DelayAsync` must have a return type of `Task<int>`. Because the return type is `Task<int>`, the evaluation of the `await` expression in `DoSomethingAsync` produces an integer as the following statement demonstrates: `int result = await delayTask`.

The `Main` method is an example of an async method that has a return type of Task. It goes to the `DoSomethingAsync` method, and because it is expressed with a single line, it can omit the `async` and `await` keywords. Because `DoSomethingAsync` is an async method, the task for the call to `DoSomethingAsync` must be awaited, as the following statement shows: `await DoSomethingAsync();`.

C#

```
class Program
{
    static Task Main() => DoSomethingAsync();
```

```
    static async Task DoSomethingAsync()
    {
        Task<int> delayTask = DelayAsync();
        int result = await delayTask;

        // The previous two statements may be combined into
        // the following statement.
        //int result = await DelayAsync();

        Console.WriteLine($"Result: {result}");
    }

    static async Task<int> DelayAsync()
    {
        await Task.Delay(100);
        return 5;
    }
}
// Example output:
//   Result: 5
```

An async method can't declare any ref or out parameters, but it can call methods that have such parameters.

For more information about async methods, see Asynchronous programming with async and await and Async return types.

Expression body definitions

It is common to have method definitions that simply return immediately with the result of an expression, or that have a single statement as the body of the method. There is a syntax shortcut for defining such methods using `=>`:

C#

```
public Point Move(int dx, int dy) => new Point(x + dx, y + dy);
public void Print() => Console.WriteLine(First + " " + Last);
// Works with operators, properties, and indexers too.
public static Complex operator +(Complex a, Complex b) => a.Add(b);
public string Name => First + " " + Last;
public Customer this[long id] => store.LookupCustomer(id);
```

If the method returns `void` or is an async method, then the body of the method must be a statement expression (same as with lambdas). For properties and indexers, they must be read only, and you don't use the `get` accessor keyword.

Iterators

An iterator performs a custom iteration over a collection, such as a list or an array. An iterator 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 when the iterator is called the next time.

You call an iterator from client code by using a foreach statement.

The return type of an iterator can be IEnumerable, IEnumerable<T>, IAsyncEnumerable<T>, IEnumerator, or IEnumerator<T>.

For more information, see Iterators.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- The C# type system
- Access Modifiers
- Static Classes and Static Class Members
- Inheritance
- Abstract and Sealed Classes and Class Members
- params
- out
- ref
- Method Parameters

Local functions (C# Programming Guide)

Article • 11/22/2024

Local functions are methods of a type that are nested in another member. They can only be called from their containing member. Local functions can be declared in and called from:

- Methods, especially iterator methods and async methods
- Constructors
- Property accessors
- Event accessors
- Anonymous methods
- Lambda expressions
- Finalizers
- Other local functions

However, local functions can't be declared inside an expression-bodied member.

> **Note**
>
> In some cases, you can use a lambda expression to implement functionality also supported by a local function. For a comparison, see Local functions vs. lambda expressions.

Local functions make the intent of your code clear. Anyone reading your code can see that the method isn't callable except by the containing method. For team projects, they also make it impossible for another developer to mistakenly call the method directly from elsewhere in the class or struct.

Local function syntax

A local function is defined as a nested method inside a containing member. Its definition has the following syntax:

```
C#
<modifiers> <return-type> <method-name> <parameter-list>
```

> **Note**
>
> The `<parameter-list>` shouldn't contain the parameters named with contextual keyword `value`. The compiler creates the temporary variable "value", which contains the referenced outer variables, which later causes ambiguity and may also cause an unexpected behaviour.

You can use the following modifiers with a local function:

- async
- unsafe
- static A static local function can't capture local variables or instance state.
- extern An external local function must be `static`.

All local variables that are defined in the containing member, including its method parameters, are accessible in a non-static local function.

Unlike a method definition, a local function definition can't include the member access modifier. Because all local functions are private, including an access modifier, such as the `private` keyword, generates compiler error CS0106, "The modifier 'private' isn't valid for this item."

The following example defines a local function named `AppendPathSeparator` that is private to a method named `GetText`:

C#

```
private static string GetText(string path, string filename)
{
     var reader = File.OpenText($"{AppendPathSeparator(path)}{filename}");
     var text = reader.ReadToEnd();
     return text;

     string AppendPathSeparator(string filepath)
     {
        return filepath.EndsWith(@"\") ? filepath : filepath + @"\";
     }
}
```

You can apply attributes to a local function, its parameters, and type parameters, as the following example shows:

C#

```
#nullable enable
private static void Process(string?[] lines, string mark)
{
    foreach (var line in lines)
    {
        if (IsValid(line))
        {
            // Processing logic...
        }
    }

    bool IsValid([NotNullWhen(true)] string? line)
    {
        return !string.IsNullOrEmpty(line) && line.Length >= mark.Length;
    }
}
```

The preceding example uses a special attribute to assist the compiler in static analysis in a nullable context.

Local functions and exceptions

One of the useful features of local functions is that they can allow exceptions to surface immediately. For iterator methods, exceptions are surfaced only when the returned sequence is enumerated, and not when the iterator is retrieved. For async methods, any exceptions thrown in an async method are observed when the returned task is awaited.

The following example defines an `OddSequence` method that enumerates odd numbers in a specified range. Because it passes a number greater than 100 to the `OddSequence` enumerator method, the method throws an ArgumentOutOfRangeException. As the output from the example shows, the exception surfaces only when you iterate the numbers, and not when you retrieve the enumerator.

C#

```
public class IteratorWithoutLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)  // line 11
      {
         Console.Write($"{x} ");
      }
   }
```

```
   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      for (int i = start; i <= end; i++)
      {
         if (i % 2 == 1)
            yield return i;
      }
   }
}
// The example displays the output like this:
//
//    Retrieved enumerator...
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithoutLocalExample.OddSequence(Int32 start, Int32
end)+MoveNext() in IteratorWithoutLocal.cs:line 22
//    at IteratorWithoutLocalExample.Main() in IteratorWithoutLocal.cs:line
11
```

If you put iterator logic into a local function, argument validation exceptions are thrown when you retrieve the enumerator, as the following example shows:

C#

```
public class IteratorWithLocalExample
{
   public static void Main()
   {
      IEnumerable<int> xs = OddSequence(50, 110);  // line 8
      Console.WriteLine("Retrieved enumerator...");

      foreach (var x in xs)
      {
         Console.Write($"{x} ");
      }
   }

   public static IEnumerable<int> OddSequence(int start, int end)
   {
      if (start < 0 || start > 99)
         throw new ArgumentOutOfRangeException(nameof(start), "start must be
between 0 and 99.");
      if (end > 100)
```

```
         throw new ArgumentOutOfRangeException(nameof(end), "end must be
less than or equal to 100.");
      if (start >= end)
         throw new ArgumentException("start must be less than end.");

      return GetOddSequenceEnumerator();

      IEnumerable<int> GetOddSequenceEnumerator()
      {
         for (int i = start; i <= end; i++)
         {
            if (i % 2 == 1)
               yield return i;
         }
      }
   }
}
// The example displays the output like this:
//
//    Unhandled exception. System.ArgumentOutOfRangeException: end must be
less than or equal to 100. (Parameter 'end')
//    at IteratorWithLocalExample.OddSequence(Int32 start, Int32 end) in
IteratorWithLocal.cs:line 22
//    at IteratorWithLocalExample.Main() in IteratorWithLocal.cs:line 8
```

Local functions vs. lambda expressions

At first glance, local functions and lambda expressions are similar. In many cases, the choice between using lambda expressions and local functions is a matter of style and personal preference. However, there are real differences in where you can use one or the other that you should be aware of.

Let's examine the differences between the local function and lambda expression implementations of the factorial algorithm. Here's the version using a local function:

C#

```
public static int LocalFunctionFactorial(int n)
{
    return nthFactorial(n);

    int nthFactorial(int number) => number < 2
        ? 1
        : number * nthFactorial(number - 1);
}
```

This version uses lambda expressions:

C#

```
public static int LambdaFactorial(int n)
{
    Func<int, int> nthFactorial = default(Func<int, int>);

    nthFactorial = number => number < 2
        ? 1
        : number * nthFactorial(number - 1);

    return nthFactorial(n);
}
```

Naming

Local functions are explicitly named like methods. Lambda expressions are anonymous methods and need to be assigned to variables of a `delegate` type, typically either `Action` or `Func` types. When you declare a local function, the process is like writing a normal method; you declare a return type and a function signature.

Function signatures and lambda expression types

Lambda expressions rely on the type of the `Action`/`Func` variable that they're assigned to determine the argument and return types. In local functions, since the syntax is much like writing a normal method, argument types and return type are already part of the function declaration.

Some lambda expressions have a *natural type*, which enables the compiler to infer the return type and parameter types of the lambda expression.

Definite assignment

Lambda expressions are objects that are declared and assigned at run time. In order for a lambda expression to be used, it needs to be definitely assigned: the `Action`/`Func` variable that it's assigned to must be declared and the lambda expression assigned to it. Notice that `LambdaFactorial` must declare and initialize the lambda expression `nthFactorial` before defining it. Not doing so results in a compile time error for referencing `nthFactorial` before assigning it.

Local functions are defined at compile time. As they're not assigned to variables, they can be referenced from any code location **where it is in scope**; in the first example `LocalFunctionFactorial`, you could declare the local function either before or after the `return` statement and not trigger any compiler errors.

These differences mean that recursive algorithms are easier to create using local functions. You can declare and define a local function that calls itself. Lambda expressions must be declared and assigned a default value before they can be reassigned to a body that references the same lambda expression.

Implementation as a delegate

Lambda expressions are converted to delegates when they're declared. Local functions are more flexible in that they can be written like a traditional method *or* as a delegate. Local functions are only converted to delegates when ***used*** as a delegate.

If you declare a local function and only reference it by calling it like a method, it won't be converted to a delegate.

Variable capture

The rules of definite assignment also affect any variables captured by the local function or lambda expression. The compiler can perform static analysis that enables local functions to definitely assign captured variables in the enclosing scope. Consider this example:

C#

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

The compiler can determine that `LocalFunction` definitely assigns `y` when called. Because `LocalFunction` is called before the `return` statement, `y` is definitely assigned at the `return` statement.

When a local function captures variables in the enclosing scope, the local function is implemented using a closure, like delegate types are.

Heap allocations

Depending on their use, local functions can avoid heap allocations that are always necessary for lambda expressions. If a local function is never converted to a delegate,

and none of the variables captured by the local function are captured by other lambdas or local functions that are converted to delegates, the compiler can avoid heap allocations.

Consider this async example:

C#

```
public async Task<string> PerformLongRunningWorkLambda(string address, int
index, string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    Func<Task<string>> longRunningWorkImplementation = async () =>
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    };

    return await longRunningWorkImplementation();
}
```

The closure for this lambda expression contains the `address`, `index`, and `name` variables. For local functions, the object that implements the closure can be a `struct` type. That struct type would be passed by reference to the local function. This difference in implementation would save on an allocation.

The instantiation necessary for lambda expressions means extra memory allocations, which might be a performance factor in time-critical code paths. Local functions don't incur this overhead.

If you know that your local function won't be converted to a delegate and none of the variables captured by it are captured by other lambdas or local functions that are converted to delegates, you can guarantee that your local function avoids being allocated on the heap by declaring it as a `static` local function.

> **Tip**

Enable .NET code style rule **IDE0062** to ensure that local functions are always marked `static`.

> **Note**
>
> The local function equivalent of this method also uses a class for the closure. Whether the closure for a local function is implemented as a `class` or a `struct` is an implementation detail. A local function may use a `struct` whereas a lambda will always use a `class`.

C#

```
public async Task<string> PerformLongRunningWork(string address, int index,
string name)
{
    if (string.IsNullOrWhiteSpace(address))
        throw new ArgumentException(message: "An address is required",
paramName: nameof(address));
    if (index < 0)
        throw new ArgumentOutOfRangeException(paramName: nameof(index),
message: "The index must be non-negative");
    if (string.IsNullOrWhiteSpace(name))
        throw new ArgumentException(message: "You must supply a name",
paramName: nameof(name));

    return await longRunningWorkImplementation();

    async Task<string> longRunningWorkImplementation()
    {
        var interimResult = await FirstWork(address);
        var secondResult = await SecondStep(index, name);
        return $"The results are {interimResult} and {secondResult}.
Enjoy.";
    }
}
```

Usage of the `yield` keyword

One final advantage not demonstrated in this sample is that local functions can be implemented as iterators, using the `yield return` syntax to produce a sequence of values.

C#

```
public IEnumerable<string> SequenceToLowercase(IEnumerable<string> input)
{
    if (!input.Any())
    {
        throw new ArgumentException("There are no items to convert to
lowercase.");
    }

    return LowercaseIterator();

    IEnumerable<string> LowercaseIterator()
    {
        foreach (var output in input.Select(item => item.ToLower()))
        {
            yield return output;
        }
    }
}
```

The `yield return` statement isn't allowed in lambda expressions. For more information, see compiler error CS1621.

While local functions can seem redundant to lambda expressions, they actually serve different purposes and have different uses. Local functions are more efficient for the case when you want to write a function that is called only from the context of another method.

C# language specification

For more information, see the Local function declarations section of the C# language specification.

See also

- Use local function instead of lambda (style rule IDE0039)
- Methods

Implicitly typed local variables (C# Programming Guide)

Article • 03/14/2023

Local variables can be declared without giving an explicit type. The `var` keyword instructs the compiler to infer the type of the variable from the expression on the right side of the initialization statement. The inferred type may be a built-in type, an anonymous type, a user-defined type, or a type defined in the .NET class library. For more information about how to initialize arrays with `var`, see Implicitly Typed Arrays.

The following examples show various ways in which local variables can be declared with `var`:

C#

```
// i is compiled as an int
var i = 5;

// s is compiled as a string
var s = "Hello";

// a is compiled as int[]
var a = new[] { 0, 1, 2 };

// expr is compiled as IEnumerable<Customer>
// or perhaps IQueryable<Customer>
var expr =
    from c in customers
    where c.City == "London"
    select c;

// anon is compiled as an anonymous type
var anon = new { Name = "Terry", Age = 34 };

// list is compiled as List<int>
var list = new List<int>();
```

It is important to understand that the `var` keyword does not mean "variant" and does not indicate that the variable is loosely typed, or late-bound. It just means that the compiler determines and assigns the most appropriate type.

The `var` keyword may be used in the following contexts:

- On local variables (variables declared at method scope) as shown in the previous example.

- In a for initialization statement.

```
C#

for (var x = 1; x < 10; x++)
```

- In a foreach initialization statement.

```
C#

foreach (var item in list) {...}
```

- In a using statement.

```
C#

using (var file = new StreamReader("C:\\myfile.txt")) {...}
```

For more information, see How to use implicitly typed local variables and arrays in a query expression.

var and anonymous types

In many cases the use of `var` is optional and is just a syntactic convenience. However, when a variable is initialized with an anonymous type you must declare the variable as `var` if you need to access the properties of the object at a later point. This is a common scenario in LINQ query expressions. For more information, see Anonymous Types.

From the perspective of your source code, an anonymous type has no name. Therefore, if a query variable has been initialized with `var`, then the only way to access the properties in the returned sequence of objects is to use `var` as the type of the iteration variable in the `foreach` statement.

```
C#

class ImplicitlyTypedLocals2
{
    static void Main()
    {
        string[] words = { "aPPLE", "BlUeBeRrY", "cHeRry" };

        // If a query produces a sequence of anonymous types,
        // then use var in the foreach statement to access the properties.
        var upperLowerWords =
```

```
                from w in words
                select new { Upper = w.ToUpper(), Lower = w.ToLower() };

            // Execute the query
            foreach (var ul in upperLowerWords)
            {
                Console.WriteLine("Uppercase: {0}, Lowercase: {1}", ul.Upper,
ul.Lower);
            }
        }
}
/* Outputs:
    Uppercase: APPLE, Lowercase: apple
    Uppercase: BLUEBERRY, Lowercase: blueberry
    Uppercase: CHERRY, Lowercase: cherry
 */
```

Remarks

The following restrictions apply to implicitly-typed variable declarations:

- `var` can only be used when a local variable is declared and initialized in the same statement; the variable cannot be initialized to null, or to a method group or an anonymous function.
- `var` cannot be used on fields at class scope.
- Variables declared by using `var` cannot be used in the initialization expression. In other words, this expression is legal: `int i = (i = 20);` but this expression produces a compile-time error: `var i = (i = 20);`
- Multiple implicitly-typed variables cannot be initialized in the same statement.
- If a type named `var` is in scope, then the `var` keyword will resolve to that type name and will not be treated as part of an implicitly typed local variable declaration.

Implicit typing with the `var` keyword can only be applied to variables at local method scope. Implicit typing is not available for class fields as the C# compiler would encounter a logical paradox as it processed the code: the compiler needs to know the type of the field, but it cannot determine the type until the assignment expression is analyzed, and the expression cannot be evaluated without knowing the type. Consider the following code:

```
C#
```

```
private var bookTitles;
```

`bookTitles` is a class field given the type `var`. As the field has no expression to evaluate, it is impossible for the compiler to infer what type `bookTitles` is supposed to be. In addition, adding an expression to the field (like you would for a local variable) is also insufficient:

C#

```
private var bookTitles = new List<string>();
```

When the compiler encounters fields during code compilation, it records each field's type before processing any expressions associated with it. The compiler encounters the same paradox trying to parse `bookTitles`: it needs to know the type of the field, but the compiler would normally determine `var`'s type by analyzing the expression, which isn't possible without knowing the type beforehand.

You may find that `var` can also be useful with query expressions in which the exact constructed type of the query variable is difficult to determine. This can occur with grouping and ordering operations.

The `var` keyword can also be useful when the specific type of the variable is tedious to type on the keyboard, or is obvious, or does not add to the readability of the code. One example where `var` is helpful in this manner is with nested generic types such as those used with group operations. In the following query, the type of the query variable is `IEnumerable<IGrouping<string, Student>>`. As long as you and others who must maintain your code understand this, there is no problem with using implicit typing for convenience and brevity.

C#

```
// Same as previous example except we use the entire last name as a key.
// Query variable is an IEnumerable<IGrouping<string, Student>>
var studentQuery3 =
    from student in students
    group student by student.Last;
```

The use of `var` helps simplify your code, but its use should be restricted to cases where it is required, or when it makes your code easier to read. For more information about when to use `var` properly, see the Implicitly typed local variables section on the C# Coding Guidelines article.

See also

- C# Reference
- Implicitly Typed Arrays
- How to use implicitly typed local variables and arrays in a query expression
- Anonymous Types
- Object and Collection Initializers
- var
- LINQ in C#
- LINQ (Language-Integrated Query)
- Iteration statements
- using statement

How to use implicitly typed local variables and arrays in a query expression (C# Programming Guide)

You can use implicitly typed local variables whenever you want the compiler to determine the type of a local variable. You must use implicitly typed local variables to store anonymous types, which are often used in query expressions. The following examples illustrate both optional and required uses of implicitly typed local variables in queries.

Implicitly typed local variables are declared by using the var contextual keyword. For more information, see Implicitly Typed Local Variables and Implicitly Typed Arrays.

Examples

The following example shows a common scenario in which the `var` keyword is required: a query expression that produces a sequence of anonymous types. In this scenario, both the query variable and the iteration variable in the `foreach` statement must be implicitly typed by using `var` because you do not have access to a type name for the anonymous type. For more information about anonymous types, see Anonymous Types.

C#

```
private static void QueryNames(char firstLetter)
{
    // Create the query. Use of var is required because
    // the query produces a sequence of anonymous types:
    // System.Collections.Generic.IEnumerable<????>.
    var studentQuery =
        from student in students
        where student.FirstName[0] == firstLetter
        select new { student.FirstName, student.LastName };

    // Execute the query and display the results.
    foreach (var anonType in studentQuery)
    {
        Console.WriteLine("First = {0}, Last = {1}", anonType.FirstName,
anonType.LastName);
    }
}
```

The following example uses the `var` keyword in a situation that is similar, but in which the use of `var` is optional. Because `student.LastName` is a string, execution of the query returns a sequence of strings. Therefore, the type of `queryId` could be declared as

`System.Collections.Generic.IEnumerable<string>` instead of `var`. Keyword `var` is used for convenience. In the example, the iteration variable in the `foreach` statement is explicitly typed as a string, but it could instead be declared by using `var`. Because the type of the iteration variable is not an anonymous type, the use of `var` is an option, not a requirement. Remember, `var` itself is not a type, but an instruction to the compiler to infer and assign the type.

C#

```
// Variable queryId could be declared by using
// System.Collections.Generic.IEnumerable<string>
// instead of var.
var queryId =
    from student in students
    where student.Id > 111
    select student.LastName;

// Variable str could be declared by using var instead of string.
foreach (string str in queryId)
{
    Console.WriteLine($"Last name: {str}");
}
```

See also

- Extension members
- LINQ (Language-Integrated Query)
- LINQ in C#

Last updated on 12/02/2025

How to implement and call a custom extension method (C# Programming Guide)

This article shows how to implement your own extension methods for any .NET type. Client code can use your extension methods. Client projects must reference the assembly that contains them. Client projects must add a using directive that specifies the namespace in which the extension methods are defined.

Beginning with C# 14, there are two syntaxes you can use to define extension methods. C# 14 adds extension blocks, where you define extension members for a type. Before C# 14, you add the this modifier to the first parameter of a static method to indicate that the method appears as a member of an instance of the parameter type.

Extension blocks (C# 14 and later)

Beginning with C# 14, you can declare extension methods using *extension blocks*. An extension block is a block in a non-nested, non-generic, static class that contains extension members for a type or an instance of that type.

To define and call an extension method using extension blocks:

1. Define a static class to contain the extension method. The class can't be nested inside another type and must be visible to client code. For more information about accessibility rules, see Access Modifiers.
2. Use an extension block to declare extension members for a type.
3. In the calling code, add a `using` directive to specify the namespace that contains the extension method class.
4. Call the methods as instance methods on the type.

The following example implements an extension method named `WordCount` using the C# 14 extension block syntax. The method operates on the String class. The `CustomExtensions` namespace is imported into the application namespace, and the method is called as if it were an instance method on the string.

C#

```
using CustomExtensions;

string s = "The quick brown fox jumped over the lazy dog.";
// Call the method as if it were an
// instance method on the type. Note that the first
```

```
// parameter is not specified by the calling code.
int i = s.WordCount();
System.Console.WriteLine($"Word count of s is {i}");


namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        extension(string str)
        {
            // This is the extension member.
            // The `str` parameter is declared on the extension declaration.
            public int WordCount()
            {
                return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
            }
        }
    }
}
```

Extension methods (earlier versions)

Before C# 14, you declare an extension method by adding the this modifier to the first parameter of a static method.

To define and call an extension method using the classic syntax:

1. Define a static class to contain the extension method. The class can't be nested inside another type and must be visible to client code. For more information about accessibility rules, see Access Modifiers.
2. Implement the extension method as a static method with at least the same visibility as the containing class.
3. The first parameter of the method specifies the type that the method operates on; it must be preceded with the this modifier.
4. In the calling code, add a `using` directive to specify the namespace that contains the extension method class.
5. Call the methods as instance methods on the type.

The following example implements an extension method named `WordCount` using the classic syntax. The method operates on the String class, which is specified as the first method parameter with the `this` modifier. The `CustomExtensions` namespace is imported into the application namespace, and the method is called as if it were an instance method on the string.

C#

```
using CustomExtensions;

string s = "The quick brown fox jumped over the lazy dog.";
// Call the method as if it were an
// instance method on the type. Note that the first
// parameter is not specified by the calling code.
int i = s.WordCount();
System.Console.WriteLine($"Word count of s is {i}");


namespace CustomExtensions
{
    // Extension methods must be defined in a static class.
    public static class StringExtension
    {
        // This is the extension method.
        // The first parameter takes the "this" modifier
        // and specifies the type to be extended.
        public static int WordCount(this string str)
        {
            return str.Split(new char[] { ' ', '.', '?' },
StringSplitOptions.RemoveEmptyEntries).Length;
        }
    }
}
```

> **Note**
>
> The first parameter is not specified by calling code because it represents the type on which the method is being applied, and the compiler already knows the type of your object. You only have to provide arguments for parameters 2 through `n`.

General information

Overload resolution prefers instance or static methods defined by the type itself to extension methods. Extension methods can't access any private data in the extended class.

See also

- Extension Methods
- LINQ (Language-Integrated Query)
- Static Classes and Static Class Members
- protected
- internal

- public
- this
- namespace

Last updated on 01/28/2026

How to create a new method for an enumeration (C# Programming Guide)

You can use extension methods to add functionality specific to a particular enum type. In the following example, the `Grades` enumeration represents the possible letter grades that a student might receive in a class. An extension method named `Passing` is added to the `Grades` type so that each instance of that type now "knows" whether it represents a passing grade or not.

C#

```
public enum Grades
{
    F = 0,
    D = 1,
    C = 2,
    B = 3,
    A = 4
};

// Define an extension method in a non-nested static class.
public static class Extensions
{
    public static bool Passing(this Grades grade, Grades minPassing = Grades.D) =>
        grade >= minPassing;
}
```

You can call the extension method as though it was declared on the `enum` type:

C#

```
Grades g1 = Grades.D;
Grades g2 = Grades.F;
Console.WriteLine($"First {(g1.Passing() ? "is" : "is not")} a passing grade.");
Console.WriteLine($"Second {(g2.Passing() ? "is" : "is not")} a passing grade.");

Console.WriteLine("\r\nRaising the bar!\r\n");
Console.WriteLine($"First {(g1.Passing(Grades.C) ? "is" : "is not")} a passing
grade.");
Console.WriteLine($"Second {(g2.Passing(Grades.C) ? "is" : "is not")} a passing
grade.");
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
```

```
    Second is not a passing grade.
*/
```

Beginning with C# 14, you can declare *extension members* in an extension block. The new syntax enables you to add *extension properties*. You can also add extension members that appear to be new static methods or properties. You're no longer limited to extensions that appear to be instance methods. The following example shows an extension block that adds an instance extension property for `Passing`, and a static extension property for `MinimumPassingGrade`:

C#

```
public static class EnumExtensions
{
    private static Grades minimumPassingGrade = Grades.D;

    extension(Grades grade)
    {
        public static Grades MinimumPassingGrade
        {
            get => minimumPassingGrade;
            set => minimumPassingGrade = value;
        }

        public bool Passing => grade >= minimumPassingGrade;
    }
}
```

You call these new extension properties as though they're declared on the extended type:

C#

```
Grades g1 = Grades.D;
Grades g2 = Grades.F;
Console.WriteLine($"First {(g1.Passing ? "is" : "is not")} a passing grade.");
Console.WriteLine($"Second {(g2.Passing ? "is" : "is not")} a passing grade.");

Grades.MinimumPassingGrade = Grades.C;
Console.WriteLine($"\r\nRaising the bar. Passing grade is now
{Grades.MinimumPassingGrade}!\r\n");
Console.WriteLine($"First {(g1.Passing ? "is" : "is not")} a passing grade.");
Console.WriteLine($"Second {(g2.Passing ? "is" : "is not")} a passing grade.");
/* Output:
    First is a passing grade.
    Second is not a passing grade.

    Raising the bar!

    First is not a passing grade.
```

```
    Second is not a passing grade.
*/
```

You can learn more about the new extension members in the article on extension members and in the language reference article on the extension keyword.

See also

- Extension Methods

Last updated on 11/25/2025

Named and Optional Arguments (C# Programming Guide)

10/10/2025

Named arguments enable you to specify an argument for a parameter by matching the argument with its name rather than with its position in the parameter list. *Optional arguments* enable you to omit arguments for some parameters. Both techniques can be used with methods, indexers, constructors, and delegates.

When you use named and optional arguments, the arguments are evaluated in the order in which they appear in the argument list, not the parameter list.

Named and optional parameters enable you to supply arguments for selected parameters. This capability greatly eases calls to COM interfaces such as the Microsoft Office Automation APIs.

Named arguments

Named arguments free you from matching the order of arguments to the order of parameters in the parameter lists of called methods. The argument for each parameter is specified by parameter name. For example, a function that prints order details (such as seller name, order number, and product name) is called by sending arguments by position, in the order defined by the function.

C#

```
PrintOrderDetails("Gift Shop", 31, "Red Mug");
```

If you don't remember the order of the parameters but know their names, you can send the arguments in any order.

C#

```
PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift Shop");
PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop", orderNum: 31);
```

Named arguments also improve the readability of your code by identifying what each argument represents. In the following example method, the `sellerName` can't be null or white space. As both `sellerName` and `productName` are string types, instead of sending arguments by position, it makes sense to use named arguments to disambiguate the two and reduce confusion for anyone reading the code.

Named arguments, when used with positional arguments, are valid as long as

- they're not followed by any positional arguments, or

  C#

  ```
  PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
  ```

- they're used in the correct position. In the following example, the parameter `orderNum` is in the correct position but isn't explicitly named.

  C#

  ```
  PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
  ```

Positional arguments that follow any out-of-order named arguments are invalid.

C#

```
// This generates CS1738: Named argument specifications must appear after all
fixed arguments have been specified.
PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
```

Example

The following code implements the examples from this section along with some extra ones.

C#

```
class NamedExample
{
    static void Main(string[] args)
    {
        // The method can be called in the normal way, by using positional
arguments.
        PrintOrderDetails("Gift Shop", 31, "Red Mug");

        // Named arguments can be supplied for the parameters in any order.
        PrintOrderDetails(orderNum: 31, productName: "Red Mug", sellerName: "Gift
Shop");
        PrintOrderDetails(productName: "Red Mug", sellerName: "Gift Shop",
orderNum: 31);

        // Named arguments mixed with positional arguments are valid
        // as long as they are used in their correct position.
        PrintOrderDetails("Gift Shop", 31, productName: "Red Mug");
        PrintOrderDetails(sellerName: "Gift Shop", 31, productName: "Red Mug");
```

```
        PrintOrderDetails("Gift Shop", orderNum: 31, "Red Mug");

        // However, mixed arguments are invalid if used out-of-order.
        // The following statements will cause a compiler error.
        // PrintOrderDetails(productName: "Red Mug", 31, "Gift Shop");
        // PrintOrderDetails(31, sellerName: "Gift Shop", "Red Mug");
        // PrintOrderDetails(31, "Red Mug", sellerName: "Gift Shop");
    }

    static void PrintOrderDetails(string sellerName, int orderNum, string
productName)
    {
        if (string.IsNullOrWhiteSpace(sellerName))
        {
            throw new ArgumentException(message: "Seller name cannot be null or
empty.", paramName: nameof(sellerName));
        }

        Console.WriteLine($"Seller: {sellerName}, Order #: {orderNum}, Product:
{productName}");
    }
}
```

Optional arguments

The definition of a method, constructor, indexer, or delegate can specify its parameters are required or optional. Any call must provide arguments for all required parameters, but can omit arguments for optional parameters. A nullable reference type (`T?`) allows arguments to be explicitly `null` but doesn't inherently make a parameter optional.

Each optional parameter has a default value as part of its definition. If no argument is sent for that parameter, the default value is used. A default value must be one of the following types of expressions:

- a constant expression;
- an expression of the form `new ValType()`, where `ValType` is a value type, such as an enum or a struct;
- an expression of the form default(ValType), where `ValType` is a value type.

Optional parameters are defined at the end of the parameter list, after any required parameters. The caller must provide arguments for all required parameters and any optional parameters preceding those it specifies. Comma-separated gaps in the argument list aren't supported. For example, in the following code, instance method `ExampleMethod` is defined with one required and two optional parameters.

```
C#
```

```
public void ExampleMethod(int required, string optionalstr = "default string",
    int optionalint = 10)
```

The following call to `ExampleMethod` causes a compiler error, because an argument is provided for the third parameter but not for the second.

C#

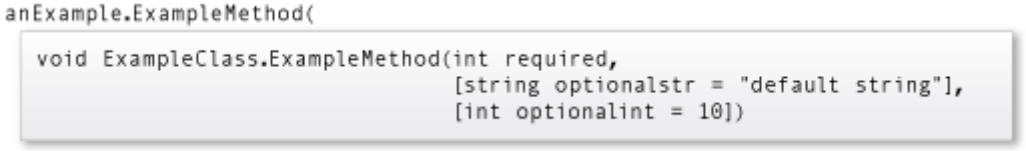
```
// anExample.ExampleMethod(3, ,4);
```

However, if you know the name of the third parameter, you can use a named argument to accomplish the task.

C#

```
anExample.ExampleMethod(3, optionalint: 4);
```

IntelliSense uses brackets to indicate optional parameters, as shown in the following illustration:



> **Note**
>
> You can also declare optional parameters by using the .NET OptionalAttribute class. `OptionalAttribute` parameters don't require a default value. However, if a default value is desired, take a look at DefaultParameterValueAttribute class.

Example

In the following example, the constructor for `ExampleClass` has one parameter, which is optional. Instance method `ExampleMethod` has one required parameter, `required`, and two optional parameters, `optionalstr` and `optionalint`. The code in `Main` shows the different ways in which the constructor and method can be invoked.

C#

```
namespace OptionalNamespace
{
```

```
    class OptionalExample
    {
        static void Main(string[] args)
        {
            // Instance anExample does not send an argument for the constructor's
            // optional parameter.
            ExampleClass anExample = new ExampleClass();
            anExample.ExampleMethod(1, "One", 1);
            anExample.ExampleMethod(2, "Two");
            anExample.ExampleMethod(3);

            // Instance anotherExample sends an argument for the constructor's
            // optional parameter.
            ExampleClass anotherExample = new ExampleClass("Provided name");
            anotherExample.ExampleMethod(1, "One", 1);
            anotherExample.ExampleMethod(2, "Two");
            anotherExample.ExampleMethod(3);

            // The following statements produce compiler errors.

            // An argument must be supplied for the first parameter, and it
            // must be an integer.
            //anExample.ExampleMethod("One", 1);
            //anExample.ExampleMethod();

            // You cannot leave a gap in the provided arguments.
            //anExample.ExampleMethod(3, ,4);
            //anExample.ExampleMethod(3, 4);

            // You can use a named parameter to make the previous
            // statement work.
            anExample.ExampleMethod(3, optionalint: 4);
        }
    }

    class ExampleClass
    {
        private string _name;

        // Because the parameter for the constructor, name, has a default
        // value assigned to it, it is optional.
        public ExampleClass(string name = "Default name")
        {
            _name = name;
        }

        // The first parameter, required, has no default value assigned
        // to it. Therefore, it is not optional. Both optionalstr and
        // optionalint have default values assigned to them. They are optional.
        public void ExampleMethod(int required, string optionalstr = "default
string",
            int optionalint = 10)
        {
            Console.WriteLine(
                $"{_name}: {required}, {optionalstr}, and {optionalint}.");
```

```
        }
    }

    // The output from this example is the following:
    // Default name: 1, One, and 1.
    // Default name: 2, Two, and 10.
    // Default name: 3, default string, and 10.
    // Provided name: 1, One, and 1.
    // Provided name: 2, Two, and 10.
    // Provided name: 3, default string, and 10.
    // Default name: 3, default string, and 4.
}
```

The preceding code illustrates several cases where optional parameters are used correctly and incorrectly. Arguments must be supplied for all required parameters. Gaps in optional arguments must be filled with named parameters.

Caller information attributes

Caller information attributes, such as CallerFilePathAttribute, CallerLineNumberAttribute, CallerMemberNameAttribute, and CallerArgumentExpressionAttribute, are used to obtain information about the caller to a method. These attributes are especially useful when you're debugging or when you need to log information about method calls.

These attributes are optional parameters with default values provided by the compiler. The caller shouldn't explicitly provide a value for these parameters.

COM interfaces

Named and optional arguments, along with support for dynamic objects, greatly improve interoperability with COM APIs, such as Office Automation APIs.

For example, the AutoFormat method in the Microsoft Office Excel Range interface has seven parameters, all of which are optional. These parameters are shown in the following illustration:

```
excelApp.get_Range("A1", "B4").AutoFormat(
```

```
dynamic Range.AutoFormat([Excel.XlRangeAutoFormat Format = 1],
[object Number = Type.Missing], [object Font = Type.Missing],
[object Alignment = Type.Missing], [object Border = Type.Missing],
[object Pattern = Type.Missing], [object Width = Type.Missing])
```

However, you can greatly simplify the call to `AutoFormat` by using named and optional arguments. Named and optional arguments enable you to omit the argument for an optional parameter if you don't want to change the parameter's default value. In the following call, a value is specified for only one of the seven parameters.

C#

```
var excelApp = new Microsoft.Office.Interop.Excel.Application();
excelApp.Workbooks.Add();
excelApp.Visible = true;

var myFormat =
    Microsoft.Office.Interop.Excel.XlRangeAutoFormat.xlRangeAutoFormatAccounting1;

excelApp.Range["A1", "B4"].AutoFormat( Format: myFormat );
```

For more information and examples, see How to use named and optional arguments in Office programming and How to access Office interop objects by using C# features.

Overload resolution

Use of named and optional arguments affects overload resolution in the following ways:

- A method, indexer, or constructor is a candidate for execution if each of its parameters either is optional or corresponds, by name or by position, to a single argument in the calling statement, and that argument can be converted to the type of the parameter.
- If more than one candidate is found, overload resolution rules for preferred conversions are applied to the arguments that are explicitly specified. Omitted arguments for optional parameters are ignored.
- If two candidates are judged to be equally good, preference goes to a candidate that doesn't have optional parameters for which arguments were omitted in the call. Overload resolution generally prefers candidates that have fewer parameters.

C# language specification

For more information, see the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Constructors (C# programming guide)

Article • 03/15/2025

A *constructor* is a method called by the runtime when an instance of a class or a struct is created. A class or struct can have multiple constructors that take different arguments. Constructors enable you to ensure that instances of the type are valid when created. For more information and examples, see Instance constructors and Using constructors.

There are several actions that are part of initializing a new instance. The following actions take place in the following order:

1. *Instance fields are set to 0*. This initialization is typically done by the runtime.
2. *Field initializers run*. The field initializers in the most derived type run.
3. *Base type field initializers run*. Field initializers starting with the direct base through each base type to System.Object.
4. *Base instance constructors run*. Any instance constructors, starting with Object.Object through each base class to the direct base class.
5. *The instance constructor runs*. The instance constructor for the type runs.
6. *Object initializers run*. If the expression includes any object initializers, they run after the instance constructor runs. Object initializers run in the textual order.

The preceding actions take place when an instance is created using the new operator. If a new instance of a `struct` is set to its `default` value, all instance fields are set to 0. Elements of an array are set to their default value of 0 or `null` when an array is created.

The static constructor, if any, runs before any of the instance constructor actions take place for any instance of the type. The static constructor runs at most once.

Beginning with C# 14, you can declare instance constructors as partial members. Partial constructors must have both a declaring and implementing declaration.

Constructor syntax

A constructor is a method with the same name as its type. Its method signature can include an optional access modifier, the method name, and its parameter list; it doesn't include a return type. The following example shows the constructor for a class named `Person`.

C#

```
public class Person
{
```

```
    private string last;
    private string first;

    public Person(string lastName, string firstName)
    {
        last = lastName;
        first = firstName;
    }

    // Remaining implementation of Person class.
}
```

If a constructor can be implemented as a single statement, you can use an expression body member. The following example defines a `Location` class whose constructor has a single string parameter, `name`. The expression body definition assigns the argument to the `locationName` field.

C#

```
public class Location
{
    private string locationName;

    public Location(string name) => Name = name;

    public string Name
    {
        get => locationName;
        set => locationName = value;
    }
}
```

If a type requires a parameter to create an instance, you can use a *primary constructor* to indicate that one or more parameters are required to instantiate the type, as shown in the following example:

C#

```
public class LabelledContainer<T>(string label)
{
    public string Label { get; } = label;
    public required T Contents
    {
        get;
        init;
    }
}
```

You can declare a primary constructor on a `partial` type. Only one primary constructor declaration is allowed. In other words, only one declaration of the `partial` type can include the parameters for the primary constructor.

Static constructors

The previous examples show instance constructors, which initialize a new object. A class or struct can also declare a static constructor, which initializes static members of the type. Static constructors are parameterless. If you don't provide a static constructor to initialize static fields, the C# compiler initializes static fields to their default value as listed in the Default values of C# types article.

The following example uses a static constructor to initialize a static field.

C#

```
public class Adult : Person
{
   private static int minimumAge;

   public Adult(string lastName, string firstName) : base(lastName,
firstName)
   { }

   static Adult() => minimumAge = 18;

   // Remaining implementation of Adult class.
}
```

You can also define a static constructor with an expression body definition, as the following example shows.

C#

```
public class Child : Person
{
   private static int maximumAge;

   public Child(string lastName, string firstName) : base(lastName,
firstName)
   { }

   static Child() => maximumAge = 18;

   // Remaining implementation of Child class.
}
```

For more information and examples, see Static Constructors.

Partial constructors

Beginning with C# 14, you can declare *partial constructors* in a partial class or struct. Any partial constructor must have a *defining declaration* and an *implementing declaration*. The signatures of the declaring and implementing partial constructors must match according to the rules of partial members. The defining declaration of the partial constructor can't use the `: base()` or `: this()` constructor initializer. You add any constructor initializer must be added on the implementing declaration.

See also

- The C# type system
- static
- Why Do Initializers Run In The Opposite Order As Constructors? Part One

Use constructors (C# programming guide)

Article • 01/18/2025

When a class or struct is instantiated, the runtime calls its constructor. Constructors have the same name as the class or struct, and they usually initialize the data members of the new object.

In the following example, a class named `Taxi` is defined by using a simple constructor. This class is then instantiated with the new operator. The runtime invokes the `Taxi` constructor immediately after memory is allocated for the new object.

C#

```
public class Taxi
{
    private string taxiTag;

    public Taxi(string tag) => taxiTag = tag;

    public override string ToString() => $"Taxi: {taxiTag}";
}

class TestTaxi
{
    static void Main()
    {
        Taxi t = new Taxi("Tag1345");
        Console.WriteLine(t);
    }
}
```

A constructor that takes no parameters is called a *parameterless constructor*. The runtime invokes the parameterless constructor when an object is instantiated using the `new` operator and no arguments are provided to `new`. C# 12 introduced *primary constructors*. A primary constructor specifies parameters that must be provided to initialize a new object. For more information, see Instance Constructors.

Unless the class is static, classes without constructors are given a public parameterless constructor by the C# compiler in order to enable class instantiation. For more information, see Static Classes and Static Class Members.

You can prevent a class from being instantiated by making the constructor private, as follows:

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

For more information, see Private Constructors.

Constructors for struct types resemble class constructors. When a struct type is instantiated with `new`, the runtime invokes a constructor. When a `struct` is set to its `default` value, the runtime initializes all memory in the struct to 0. If you declare any field initializers in a `struct` type, you must supply a parameterless constructor. For more information, see the Struct initialization and default values section of the Structure types article.

The following code uses the parameterless constructor for Int32, so that you're assured that the integer is initialized:

C#

```
int i = new int();
Console.WriteLine(i);
```

The following code, however, causes a compiler error because it doesn't use `new`, and because it tries to use an object that isn't initialized:

C#

```
int i;
Console.WriteLine(i);
```

Alternatively, some `struct` types (including all built-in numeric types) can be initialized or assigned and then used as in the following example:

C#

```
int a = 44;  // Initialize the value type...
int b;
b = 33;      // Or assign it before using it.
Console.WriteLine("{0}, {1}", a, b);
```

Both classes and structs can define constructors that take parameters, including primary constructors. Constructors that take parameters must be called through a `new` statement or a base statement. Classes and structs can also define multiple constructors, and neither is required to define a parameterless constructor. For example:

```
C#

public class Employee
{
    public int Salary;

    public Employee() { }

    public Employee(int annualSalary) => Salary = annualSalary;

    public Employee(int weeklySalary, int numberOfWeeks) => Salary =
weeklySalary * numberOfWeeks;
}
```

This class can be created by using either of the following statements:

```
C#

Employee e1 = new Employee(30000);
Employee e2 = new Employee(500, 52);
```

A constructor can use the `base` keyword to call the constructor of a base class. For example:

```
C#

public class Manager : Employee
{
    public Manager(int annualSalary)
        : base(annualSalary)
    {
        //Add further instructions here.
    }
}
```

In this example, the constructor for the base class is called before the block for the constructor executes. The `base` keyword can be used with or without parameters. Any parameters to the constructor can be used as parameters to `base`, or as part of an expression. For more information, see base.

In a derived class, if a base-class constructor isn't called explicitly by using the `base` keyword, the parameterless constructor, if there's one, is called implicitly. The following constructor declarations are effectively the same:

C#

```
public Manager(int initialData)
{
    //Add further instructions here.
}
```

C#

```
public Manager(int initialData)
    : base()
{
    //Add further instructions here.
}
```

If a base class doesn't offer a parameterless constructor, the derived class must make an explicit call to a base constructor by using `base`.

A constructor can invoke another constructor in the same object by using the this keyword. Like `base`, `this` can be used with or without parameters, and any parameters in the constructor are available as parameters to `this`, or as part of an expression. For example, the second constructor in the previous example can be rewritten using `this`:

C#

```
public Employee(int weeklySalary, int numberOfWeeks)
    : this(weeklySalary * numberOfWeeks)
{
}
```

The use of the `this` keyword in the previous example causes the following constructor to be called:

C#

```
public Employee(int annualSalary) => Salary = annualSalary;
```

Constructors can be marked as public, private, protected, internal, protected internal or private protected. These access modifiers define how users of the class can construct the class. For more information, see Access Modifiers.

A constructor can be declared static by using the static keyword. Static constructors are called automatically, before any static fields are accessed, and are used to initialize static class members. For more information, see Static Constructors.

C# Language Specification

For more information, see Instance constructors and Static constructors in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- The C# type system
- Constructors
- Finalizers

Instance constructors (C# programming guide)

Article • 05/31/2024

You declare an instance constructor to specify the code that is executed when you create a new instance of a type with the new expression. To initialize a static class or static variables in a nonstatic class, you can define a static constructor.

As the following example shows, you can declare several instance constructors in one type:

C#

```
class Coords
{
    public Coords()
        : this(0, 0)
    {  }

    public Coords(int x, int y)
    {
        X = x;
        Y = y;
    }

    public int X { get; set; }
    public int Y { get; set; }

    public override string ToString() => $"({X},{Y})";
}

class Example
{
    static void Main()
    {
        var p1 = new Coords();
        Console.WriteLine($"Coords #1 at {p1}");
        // Output: Coords #1 at (0,0)

        var p2 = new Coords(5, 3);
        Console.WriteLine($"Coords #2 at {p2}");
        // Output: Coords #2 at (5,3)
    }
}
```

In the preceding example, the first, parameterless, constructor calls the second constructor with both arguments equal `0`. To do that, use the `this` keyword.

When you declare an instance constructor in a derived class, you can call a constructor of a base class. To do that, use the `base` keyword, as the following example shows:

C#

```
abstract class Shape
{
    public const double pi = Math.PI;
    protected double x, y;

    public Shape(double x, double y)
    {
        this.x = x;
        this.y = y;
    }

    public abstract double Area();
}

class Circle : Shape
{
    public Circle(double radius)
        : base(radius, 0)
    {  }

    public override double Area() => pi * x * x;
}

class Cylinder : Circle
{
    public Cylinder(double radius, double height)
        : base(radius)
    {
        y = height;
    }

    public override double Area() => (2 * base.Area()) + (2 * pi * x * y);
}

class Example
{
    static void Main()
    {
        double radius = 2.5;
        double height = 3.0;

        var ring = new Circle(radius);
        Console.WriteLine($"Area of the circle = {ring.Area():F2}");
        // Output: Area of the circle = 19.63

        var tube = new Cylinder(radius, height);
        Console.WriteLine($"Area of the cylinder = {tube.Area():F2}");
        // Output: Area of the cylinder = 86.39
```

```
    }
}
```

Parameterless constructors

If a *class* has no explicit instance constructors, C# provides a parameterless constructor that you can use to instantiate an instance of that class, as the following example shows:

C#

```
public class Person
{
    public int age;
    public string name = "unknown";
}

class Example
{
    static void Main()
    {
        var person = new Person();
        Console.WriteLine($"Name: {person.name}, Age: {person.age}");
        // Output:  Name: unknown, Age: 0
    }
}
```

That constructor initializes instance fields and properties according to the corresponding initializers. If a field or property has no initializer, its value is set to the default value of the field's or property's type. If you declare at least one instance constructor in a class, C# doesn't provide a parameterless constructor.

A *structure* type always provides a parameterless constructor. The parameterless constructor is either an implicit parameterless constructor that produces the default value of a type or an explicitly declared parameterless constructor. For more information, see the Struct initialization and default values section of the Structure types article.

Primary constructors

Beginning in C# 12, you can declare a *primary constructor* in classes and structs. You place any parameters in parentheses following the type name:

C#

```
public class NamedItem(string name)
{
    public string Name => name;
}
```

The parameters to a primary constructor are in scope in the entire body of the declaring type. They can initialize properties or fields. They can be used as variables in methods or local functions. They can be passed to a base constructor.

A primary constructor indicates that these parameters are necessary for any instance of the type. Any explicitly written constructor must use the `this(...)` initializer syntax to invoke the primary constructor. That ensures that the primary constructor parameters are definitely assigned by all constructors. For any `class` type, including `record class` types, the implicit parameterless constructor isn't emitted when a primary constructor is present. For any `struct` type, including `record struct` types, the implicit parameterless constructor is always emitted, and always initializes all fields, including primary constructor parameters, to the 0-bit pattern. If you write an explicit parameterless constructor, it must invoke the primary constructor. In that case, you can specify a different value for the primary constructor parameters. The following code shows examples of primary constructors.

C#

```
// name isn't captured in Widget.
// width, height, and depth are captured as private fields
public class Widget(string name, int width, int height, int depth) :
NamedItem(name)
{
    public Widget() : this("N/A", 1,1,1) {} // unnamed unit cube

    public int WidthInCM => width;
    public int HeightInCM => height;
    public int DepthInCM => depth;

    public int Volume => width * height * depth;
}
```

You can add attributes to the synthesized primary constructor method by specifying the `method:` target on the attribute:

C#

```
[method: MyAttribute]
public class TaggedWidget(string name)
{
```

```
    // details elided
}
```

If you don't specify the `method` target, the attribute is placed on the class rather than the method.

In `class` and `struct` types, primary constructor parameters are available anywhere in the body of the type. The parameter can be implemented as a captured private field. If the only references to a parameter are initializers and constructor calls, that parameter isn't captured in a private field. Uses in other members of the type cause the compiler to capture the parameter in a private field.

If the type includes the `record` modifier, the compiler instead synthesizes a public property with the same name as the primary constructor parameter. For `record class` types, if a primary constructor parameter uses the same name as a base primary constructor, that property is a public property of the base `record class` type. It isn't duplicated in the derived `record class` type. These properties aren't generated for non-`record` types.

See also

- Classes, structs, and records
- Constructors
- Finalizers
- base
- this
- Primary constructors feature spec

Private Constructors (C# Programming Guide)

Article • 01/12/2022

A private constructor is a special instance constructor. It is generally used in classes that contain static members only. If a class has one or more private constructors and no public constructors, other classes (except nested classes) cannot create instances of this class. For example:

C#

```
class NLog
{
    // Private Constructor:
    private NLog() { }

    public static double e = Math.E;  //2.71828...
}
```

The declaration of the empty constructor prevents the automatic generation of a parameterless constructor. Note that if you do not use an access modifier with the constructor it will still be private by default. However, the private modifier is usually used explicitly to make it clear that the class cannot be instantiated.

Private constructors are used to prevent creating instances of a class when there are no instance fields or methods, such as the Math class, or when a method is called to obtain an instance of a class. If all the methods in the class are static, consider making the complete class static. For more information see Static Classes and Static Class Members.

Example

The following is an example of a class using a private constructor.

C#

```
public class Counter
{
    private Counter() { }

    public static int currentCount;

    public static int IncrementCount()
    {
        return ++currentCount;
```

```
    }
}

class TestCounter
{
    static void Main()
    {
        // If you uncomment the following statement, it will generate
        // an error because the constructor is inaccessible:
        // Counter aCounter = new Counter();   // Error

        Counter.currentCount = 100;
        Counter.IncrementCount();
        Console.WriteLine($"New count: {Counter.currentCount}");

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: New count: 101
```

Notice that if you uncomment the following statement from the example, it will generate an error because the constructor is inaccessible because of its protection level:

C#

```
// Counter aCounter = new Counter();   // Error
```

See also

- The C# type system
- Constructors
- Finalizers
- private
- public

Static Constructors (C# Programming Guide)

Article • 08/02/2024

A static constructor is used to initialize any static data, or to perform a particular action that needs to be performed only once. It's called automatically before the first instance is created or any static members are referenced. A static constructor is called at most once.

C#

```
class SimpleClass
{
    // Static variable that must be initialized at run time.
    static readonly long baseline;

    // Static constructor is called at most one time, before any
    // instance constructor is invoked or member is accessed.
    static SimpleClass()
    {
        baseline = DateTime.Now.Ticks;
    }
}
```

There are several actions that are part of static initialization. Those actions take place in the following order:

1. *Static fields are set to 0*. The runtime typically does this initialization.
2. *Static field initializers run*. The static field initializers in the most derived type run.
3. *Base type static field initializers run*. Static field initializers starting with the direct base through each base type to System.Object.
4. *Any static constructor runs*. Any static constructors, from the ultimate base class of Object.Object through each base class through the type run. The order of static constructor execution isn't specified. However, all static constructors in the hierarchy run before any instances are created.

> **ⓘ Important**
>
> There is one important exception to the rule that a static constructor runs before any instance is created. If a static field initializer creates an instance of the type, that initializer runs (including any call to an instance constructor) before the static constructor runs. This is most common in the *singleton pattern* as shown in the following example:

C#

```
public class Singleton
{
    // Static field initializer calls instance constructor.
    private static Singleton instance = new Singleton();

    private Singleton()
    {
        Console.WriteLine("Executes before static constructor.");
    }

    static Singleton()
    {
        Console.WriteLine("Executes after instance constructor.");
    }

    public static Singleton Instance => instance;
}
```

A module initializer can be an alternative to a static constructor. For more information, see the specification for module initializers.

Remarks

Static constructors have the following properties:

- A static constructor doesn't take access modifiers or have parameters.
- A class or struct can only have one static constructor.
- Static constructors can't be inherited or overloaded.
- A static constructor can't be called directly and is only meant to be called by the common language runtime (CLR). It's invoked automatically.
- The user has no control on when the static constructor is executed in the program.
- A static constructor is called automatically. It initializes the class before the first instance is created or any static members declared in that class (not its base classes) are referenced. A static constructor runs before an instance constructor. If static field variable initializers are present in the class of the static constructor, they run in the textual order in which they appear in the class declaration. The initializers run immediately before the static constructor.
- If you don't provide a static constructor to initialize static fields, all static fields are initialized to their default value as listed in Default values of C# types.
- If a static constructor throws an exception, the runtime doesn't invoke it a second time, and the type remains uninitialized for the lifetime of the application domain. Most commonly, a TypeInitializationException exception is thrown when a static

constructor is unable to instantiate a type or for an unhandled exception occurring within a static constructor. For static constructors that aren't explicitly defined in source code, troubleshooting might require inspection of the intermediate language (IL) code.

- The presence of a static constructor prevents the addition of the BeforeFieldInit type attribute. This limits runtime optimization.
- A field declared as `static readonly` can only be assigned as part of its declaration or in a static constructor. When an explicit static constructor isn't required, initialize static fields at declaration rather than through a static constructor for better runtime optimization.
- The runtime calls a static constructor no more than once in a single application domain. That call is made in a locked region based on the specific type of the class. No extra locking mechanisms are needed in the body of a static constructor. To avoid the risk of deadlocks, don't block the current thread in static constructors and initializers. For example, don't wait on tasks, threads, wait handles or events, don't acquire locks, and don't execute blocking parallel operations such as parallel loops, `Parallel.Invoke` and Parallel LINQ queries.

> **Note**
>
> Though not directly accessible, the presence of an explicit static constructor should be documented to assist with troubleshooting initialization exceptions.

Usage

- A typical use of static constructors is when the class is using a log file and the constructor is used to write entries to this file.
- Static constructors are also useful when creating wrapper classes for unmanaged code, when the constructor can call the `LoadLibrary` method.
- Static constructors are also a convenient place to enforce run-time checks on the type parameter that can't be checked at compile time via type-parameter constraints.

Example

In this example, class `Bus` has a static constructor. When the first instance of `Bus` is created (`bus1`), the static constructor is invoked to initialize the class. The sample output verifies that the static constructor runs only one time, even though two instances of `Bus` are created, and that it runs before the instance constructor runs.

C#

```
public class Bus
{
    // Static variable used by all Bus instances.
    // Represents the time the first bus of the day starts its route.
    protected static readonly DateTime globalStartTime;

    // Property for the number of each bus.
    protected int RouteNumber { get; set; }

    // Static constructor to initialize the static variable.
    // It is invoked before the first instance constructor is run.
    static Bus()
    {
        globalStartTime = DateTime.Now;

        // The following statement produces the first line of output,
        // and the line occurs only once.
        Console.WriteLine($"Static constructor sets global start time to
{globalStartTime.ToLongTimeString()}");
    }

    // Instance constructor.
    public Bus(int routeNum)
    {
        RouteNumber = routeNum;
        Console.WriteLine($"Bus #{RouteNumber} is created.");
    }

    // Instance method.
    public void Drive()
    {
        TimeSpan elapsedTime = DateTime.Now - globalStartTime;

        // For demonstration purposes we treat milliseconds as minutes to
simulate
        // actual bus times. Do not do this in your actual bus schedule
program!
        Console.WriteLine($"{this.RouteNumber} is starting its route
{elapsedTime.Milliseconds:N2} minutes after global start time
{globalStartTime.ToShortTimeString()}.");
    }
}

class TestBus
{
    static void Main()
    {
        // The creation of this instance activates the static constructor.
        Bus bus1 = new Bus(71);

        // Create a second bus.
        Bus bus2 = new Bus(72);
```

```
        // Send bus1 on its way.
        bus1.Drive();

        // Wait for bus2 to warm up.
        System.Threading.Thread.Sleep(25);

        // Send bus2 on its way.
        bus2.Drive();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Sample output:
    Static constructor sets global start time to 3:57:08 PM.
    Bus #71 is created.
    Bus #72 is created.
    71 is starting its route 6.00 minutes after global start time 3:57 PM.
    72 is starting its route 31.00 minutes after global start time 3:57 PM.
*/
```

C# language specification

For more information, see the Static constructors section of the C# language specification.

See also

- The C# type system
- Constructors
- Static Classes and Static Class Members
- Finalizers
- Constructor Design Guidelines
- Security Warning - CA2121: Static constructors should be private
- Module initializers

How to write a copy constructor (C# Programming Guide)

Article • 03/12/2024

C# records provide a copy constructor for objects, but for classes you have to write one yourself.

> **ⓘ Important**
>
> Writing copy constructors that work for all derived types in a class hierarchy can be difficult. If your class isn't `sealed`, you should strongly consider creating a hierarchy of `record class` types to use the compiler-synthesized copy constructor.

Example

In the following example, the `Person` class defines a copy constructor that takes, as its argument, an instance of `Person`. The values of the properties of the argument are assigned to the properties of the new instance of `Person`. The code contains an alternative copy constructor that sends the `Name` and `Age` properties of the instance that you want to copy to the instance constructor of the class. The `Person` class is `sealed`, so no derived types can be declared that could introduce errors by copying only the base class.

C#

```
public sealed class Person
{
    // Copy constructor.
    public Person(Person previousPerson)
    {
        Name = previousPerson.Name;
        Age = previousPerson.Age;
    }

    //// Alternate copy constructor calls the instance constructor.
    //public Person(Person previousPerson)
    //    : this(previousPerson.Name, previousPerson.Age)
    //{
    //}

    // Instance constructor.
    public Person(string name, int age)
```

```
    {
        Name = name;
        Age = age;
    }

    public int Age { get; set; }

    public string Name { get; set; }

    public string Details()
    {
        return Name + " is " + Age.ToString();
    }
}

class TestPerson
{
    static void Main()
    {
        // Create a Person object by using the instance constructor.
        Person person1 = new Person("George", 40);

        // Create another Person object, copying person1.
        Person person2 = new Person(person1);

        // Change each person's age.
        person1.Age = 39;
        person2.Age = 41;

        // Change person2's name.
        person2.Name = "Charles";

        // Show details to verify that the name and age fields are distinct.
        Console.WriteLine(person1.Details());
        Console.WriteLine(person2.Details());

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output:
// George is 39
// Charles is 41
```

See also

- ICloneable
- Records
- The C# type system
- Constructors

- Finalizers

Extension members (C# Programming Guide)

Extension members enable you to "add" methods to existing types without creating a new derived type, recompiling, or otherwise modifying the original type.

Beginning with C# 14, there are two syntaxes you use to define extension methods. C# 14 adds extension blocks, where you define multiple extension members for a type or an instance of a type. Before C# 14, you add the this modifier to the first parameter of a static method to indicate that the method appears as a member of an instance of the parameter type.

Extension blocks support multiple member types: methods, properties, and operators. With extension blocks, you can define both instance extensions and static extensions. Instance extensions extend an instance of the type; static extensions extend the type itself. The form of extension methods declared with the `this` modifier supports instance extension methods.

Extension methods are static methods, but they're called as if they were instance methods on the extended type. For client code written in C#, F# and Visual Basic, there's no apparent difference between calling an extension method and the methods defined in a type. Both forms of extension methods are compiled to the same IL (Intermediate Language). Consumers of extension members don't need to know which syntax was used to define extension methods.

The most common extension members are the LINQ standard query operators that add query functionality to the existing System.Collections.IEnumerable and System.Collections.Generic.IEnumerable<T> types. To use the standard query operators, first bring them into scope with a `using System.Linq` directive. Then any type that implements IEnumerable<T> appears to have instance methods such as GroupBy, OrderBy, Average, and so on. You can see these extra methods in IntelliSense statement completion when you type "dot" after an instance of an IEnumerable<T> type such as List<T> or Array.

OrderBy example

The following example shows how to call the standard query operator `OrderBy` method on an array of integers. The expression in parentheses is a lambda expression. Many standard query operators take lambda expressions as parameters. For more information, see Lambda Expressions.

C#

```
int[] numbers = [10, 45, 15, 39, 21, 26];
IOrderedEnumerable<int> result = numbers.OrderBy(g => g);
```

```
foreach (int i in result)
{
    Console.Write(i + " ");
}
//Output: 10 15 21 26 39 45
```

Extension methods are defined as static methods but are called by using instance method syntax. Their first parameter specifies which type the method operates on. The parameter follows the this modifier. Extension methods are only in scope when you explicitly import the namespace into your source code with a `using` directive.

Declare extension members

Beginning with C# 14, you can declare *extension blocks*. An extension block is a block in a non-nested, nongeneric, static class that contains extension members for a type or an instance of that type. The following code example defines an extension block for the `string` type. The extension block contains one member: a method that counts the words in the string:

C#

```
namespace CustomExtensionMembers;

public static class MyExtensions
{
    extension(string str)
    {
        public int WordCount() =>
            str.Split([' ', '.', '?'],
StringSplitOptions.RemoveEmptyEntries).Length;
    }
}
```

Extension members in an extension block can use open or closed generics. The type parameters can include constraints:

C#

```
public static class MyGenericExtensions
{
    extension<T>(IEnumerable<T> source)
        where T : IEquatable<T>
    {
        public IEnumerable<T> ValuesEqualTo(T threshold)
            => source.Where(x => x.Equals(threshold));
    }
}
```

Before C# 14, you declare an extension method by adding the `this` modifier to the first parameter:

C#

```
namespace CustomExtensionMethods;

public static class MyExtensions
{
    public static int WordCount(this string str) =>
        str.Split([' ', '.', '?'], StringSplitOptions.RemoveEmptyEntries).Length;
}
```

Both forms of extensions must be defined inside a non-nested, nongeneric static class.

And it can be called from an application by using the syntax for accessing instance members:

C#

```
string s = "Hello Extension Methods";
int i = s.WordCount();
```

While extension members add new capabilities to an existing type, extension members don't violate the principle of encapsulation. The access declarations for all members of the extended type apply to extension members.

Both the `MyExtensions` class and the `WordCount` method are `static`, and it can be accessed like all other `static` members. The `WordCount` method can be invoked like other `static` methods as follows:

C#

```
string s = "Hello Extension Methods";
int i = MyExtensions.WordCount(s);
```

The preceding C# code applies to both the extension block and `this` syntax for extension members. The preceding code:

- Declares and assigns a new `string` named `s` with a value of `"Hello Extension Methods"`.
- Calls `MyExtensions.WordCount` given argument `s`.

For more information, see How to implement and call a custom extension method.

In general, you probably call extension members far more often than you implement them. Because extension members are called as though they're declared as members of the extended class, no special knowledge is required to use them from client code. To enable extension

members for a particular type, just add a `using` directive for the namespace in which the methods are defined. For example, to use the standard query operators, add this `using` directive to your code:

C#

```
using System.Linq;
```

Binding extension members at compile time

You can use extension members to extend a class or interface, but not to override behavior defined in a class. An extension member with the same name and signature as an interface or class members are never called. At compile time, extension members always have lower priority than instance (or static) members defined in the type itself. In other words, if a type has a method named `Process(int i)`, and you have an extension method with the same signature, the compiler always binds to the member method. When the compiler encounters a member invocation, it first looks for a match in the type's members. If no match is found, it searches for any extension members that are defined for the type. It binds to the first extension member that it finds. The following example demonstrates the rules that the C# compiler follows in determining whether to bind to an instance member on the type, or to an extension member. The static class `Extensions` contains extension members defined for any type that implements `IMyInterface`:

C#

```
public interface IMyInterface
{
    void MethodB();
}

// Define extension methods for IMyInterface.

// The following extension methods can be accessed by instances of any
// class that implements IMyInterface.
public static class Extension
{
    public static void MethodA(this IMyInterface myInterface, int i) =>
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, int
i)");

    public static void MethodA(this IMyInterface myInterface, string s) =>
        Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, string
s)");

    // This method is never called in ExtensionMethodsDemo1, because each
    // of the three classes A, B, and C implements a method named MethodB
```

```
    // that has a matching signature.
    public static void MethodB(this IMyInterface myInterface) =>
        Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");
}
```

The equivalent extensions can be declared using the C# 14 extension member syntax:

C#

```
public static class Extension
{
    extension(IMyInterface myInterface)
    {
        public void MethodA(int i) =>
            Console.WriteLine("Extension.MethodA(this IMyInterface myInterface, int
i)");

        public void MethodA(string s) =>
            Console.WriteLine("Extension.MethodA(this IMyInterface myInterface,
string s)");

        // This method is never called in ExtensionMethodsDemo1, because each
        // of the three classes A, B, and C implements a method named MethodB
        // that has a matching signature.
        public void MethodB() =>
            Console.WriteLine("Extension.MethodB(this IMyInterface myInterface)");
    }
}
```

Classes `A`, `B`, and `C` all implement the interface:

C#

```
// Define three classes that implement IMyInterface, and then use them to test
// the extension methods.
class A : IMyInterface
{
    public void MethodB() { Console.WriteLine("A.MethodB()"); }
}

class B : IMyInterface
{
    public void MethodB() { Console.WriteLine("B.MethodB()"); }
    public void MethodA(int i) { Console.WriteLine("B.MethodA(int i)"); }
}

class C : IMyInterface
{
    public void MethodB() { Console.WriteLine("C.MethodB()"); }
    public void MethodA(object obj)
    {
        Console.WriteLine("C.MethodA(object obj)");
```

```
    }
}
```

The `MethodB` extension method is never called because its name and signature exactly match methods already implemented by the classes. When the compiler can't find an instance method with a matching signature, it binds to a matching extension method if one exists.

C#

```
// Declare an instance of class A, class B, and class C.
A a = new A();
B b = new B();
C c = new C();

// For a, b, and c, call the following methods:
//      -- MethodA with an int argument
//      -- MethodA with a string argument
//      -- MethodB with no argument.

// A contains no MethodA, so each call to MethodA resolves to
// the extension method that has a matching signature.
a.MethodA(1);           // Extension.MethodA(IMyInterface, int)
a.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

// A has a method that matches the signature of the following call
// to MethodB.
a.MethodB();            // A.MethodB()

// B has methods that match the signatures of the following
// method calls.
b.MethodA(1);           // B.MethodA(int)
b.MethodB();            // B.MethodB()

// B has no matching method for the following call, but
// class Extension does.
b.MethodA("hello");     // Extension.MethodA(IMyInterface, string)

// C contains an instance method that matches each of the following
// method calls.
c.MethodA(1);           // C.MethodA(object)
c.MethodA("hello");     // C.MethodA(object)
c.MethodB();            // C.MethodB()
/* Output:
    Extension.MethodA(this IMyInterface myInterface, int i)
    Extension.MethodA(this IMyInterface myInterface, string s)
    A.MethodB()
    B.MethodA(int i)
    B.MethodB()
    Extension.MethodA(this IMyInterface myInterface, string s)
    C.MethodA(object obj)
    C.MethodA(object obj)
```

```
    C.MethodB()
 */
```

Common usage patterns

Collection Functionality

In the past, it was common to create "Collection Classes" that implemented the System.Collections.Generic.IEnumerable<T> interface for a given type and contained functionality that acted on collections of that type. While there's nothing wrong with creating this type of collection object, the same functionality can be achieved by using an extension on the System.Collections.Generic.IEnumerable<T>. Extensions have the advantage of allowing the functionality to be called from any collection such as an System.Array or System.Collections.Generic.List<T> that implements System.Collections.Generic.IEnumerable<T> on that type. An example of this using an Array of Int32 can be found earlier in this article.

Layer-Specific Functionality

When using an Onion Architecture or other layered application design, it's common to have a set of Domain Entities or Data Transfer Objects that can be used to communicate across application boundaries. These objects generally contain no functionality, or only minimal functionality that applies to all layers of the application. Extension methods can be used to add functionality that's specific to each application layer.

C#

```
public class DomainEntity
{
    public int Id { get; set; }
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
}

static class DomainEntityExtensions
{
    static string FullName(this DomainEntity value)
        => $"{value.FirstName} {value.LastName}";
}
```

You can declare an equivalent `FullName` property in C# 14 and later using the new extension block syntax:

C#

```
static class DomainEntityExtensions
{
    extension(DomainEntity value)
    {
        string FullName => $"{value.FirstName} {value.LastName}";
    }
}
```

Extending Predefined Types

Rather than creating new objects when reusable functionality needs to be created, you can often extend an existing type, such as a .NET or CLR type. As an example, if you don't use extension methods, you might create an `Engine` or `Query` class to do the work of executing a query on a SQL Server that might be called from multiple places in our code. However you can instead extend the System.Data.SqlClient.SqlConnection class using extension methods to perform that query from anywhere you have a connection to a SQL Server. Other examples might be to add common functionality to the System.String class, extend the data processing capabilities of the System.IO.Stream object, and System.Exception objects for specific error handling functionality. These types of use-cases are limited only by your imagination and good sense.

Extending predefined types can be difficult with `struct` types because they're passed by value to methods. That means any changes to the struct are made to a copy of the struct. Those changes aren't visible once the extension method exits. You can add the `ref` modifier to the first argument making it a `ref` extension method. The `ref` keyword can appear before or after the `this` keyword without any semantic differences. Adding the `ref` modifier indicates that the first argument is passed by reference. This technique enables you to write extension methods that change the state of the struct being extended (note that private members aren't accessible). Only value types or generic types constrained to struct (For more information about these rules, see the article on the struct constraint) are allowed as the first parameter of a `ref` extension method or as the receiver of an extension block. The following example shows how to use a `ref` extension method to directly modify a built-in type without the need to reassign the result or pass it through a function with the `ref` keyword:

C#

```
public static class IntExtensions
{
    public static void Increment(this int number)
        => number++;

    // Take note of the extra ref keyword here
```

```
    public static void RefIncrement(this ref int number)
        => number++;
}
```

The equivalent extension blocks are shown in the following code:

C#

```
public static class IntExtensions
{
    extension(int number)
    {
        public void Increment()
            => number++;
    }

    // Take note of the extra ref keyword here
    extension(ref int number)
    {
        public void RefIncrement()
            => number++;
    }
}
```

Different extension blocks are required to distinguish by-value and by-ref parameter modes for the receiver.

You can see the difference applying `ref` to the receiver has in the following example:

C#

```
int x = 1;

// Takes x by value leading to the extension method
// Increment modifying its own copy, leaving x unchanged
x.Increment();
Console.WriteLine($"x is now {x}"); // x is now 1

// Takes x by reference leading to the extension method
// RefIncrement changing the value of x directly
x.RefIncrement();
Console.WriteLine($"x is now {x}"); // x is now 2
```

You can apply the same technique by adding `ref` extension members to user-defined struct types:

C#

```
public struct Account
{
```

```
    public uint id;
    public float balance;

    private int secret;
}

public static class AccountExtensions
{
    // ref keyword can also appear before the this keyword
    public static void Deposit(ref this Account account, float amount)
    {
        account.balance += amount;

        // The following line results in an error as an extension
        // method is not allowed to access private members
        // account.secret = 1; // CS0122
    }
}
```

The preceding sample can also be created using extension blocks in C# 14:

C#

```
public static class AccountExtensions
{
    extension(ref Account account)
    {
        // ref keyword can also appear before the this keyword
        public void Deposit(float amount)
        {
            account.balance += amount;

            // The following line results in an error as an extension
            // method is not allowed to access private members
            // account.secret = 1; // CS0122
        }
    }
}
```

You can access these extension methods as follows:

C#

```
Account account = new()
{
    id = 1,
    balance = 100f
};

Console.WriteLine($"I have ${account.balance}"); // I have $100
```

```
account.Deposit(50f);
Console.WriteLine($"I have ${account.balance}"); // I have $150
```

General Guidelines

It's preferable to add functionality by modifying an object's code or deriving a new type whenever it's reasonable and possible to do so. Extension methods are a crucial option for creating reusable functionality throughout the .NET ecosystem. Extension members are preferable when the original source isn't under your control, when a derived object is inappropriate or impossible, or when the functionality has limited scope.

For more information on derived types, see Inheritance.

If you do implement extension methods for a given type, remember the following points:

- An extension method isn't called if it has the same signature as a method defined in the type.
- Extension methods are brought into scope at the namespace level. For example, if you have multiple static classes that contain extension methods in a single namespace named `Extensions`, all of them are brought into scope by the `using Extensions;` directive.

For a class library that you implemented, you shouldn't use extension methods to avoid incrementing the version number of an assembly. If you want to add significant functionality to a library for which you own the source code, follow the .NET guidelines for assembly versioning. For more information, see Assembly Versioning.

See also

- Parallel Programming Samples (many examples demonstrate extension methods)
- Lambda Expressions
- Standard Query Operators Overview

Last updated on 11/20/2025

Finalizers (C# Programming Guide)

Article • 03/14/2023

Finalizers (historically referred to as **destructors**) are used to perform any necessary final clean-up when a class instance is being collected by the garbage collector. In most cases, you can avoid writing a finalizer by using the System.Runtime.InteropServices.SafeHandle or derived classes to wrap any unmanaged handle.

Remarks

- Finalizers cannot be defined in structs. They are only used with classes.
- A class can only have one finalizer.
- Finalizers cannot be inherited or overloaded.
- Finalizers cannot be called. They are invoked automatically.
- A finalizer does not take modifiers or have parameters.

For example, the following is a declaration of a finalizer for the `Car` class.

C#

```
class Car
{
    ~Car()  // finalizer
    {
        // cleanup statements...
    }
}
```

A finalizer can also be implemented as an expression body definition, as the following example shows.

C#

```
public class Destroyer
{
   public override string ToString() => GetType().Name;

   ~Destroyer() => Console.WriteLine($"The {ToString()} finalizer is
executing.");
}
```

The finalizer implicitly calls Finalize on the base class of the object. Therefore, a call to a finalizer is implicitly translated to the following code:

C#

```
protected override void Finalize()
{
    try
    {
        // Cleanup statements...
    }
    finally
    {
        base.Finalize();
    }
}
```

This design means that the `Finalize` method is called recursively for all instances in the inheritance chain, from the most-derived to the least-derived.

> **Note**
>
> Empty finalizers should not be used. When a class contains a finalizer, an entry is created in the `Finalize` queue. This queue is processed by the garbage collector. When the GC processes the queue, it calls each finalizer. Unnecessary finalizers, including empty finalizers, finalizers that only call the base class finalizer, or finalizers that only call conditionally emitted methods, cause a needless loss of performance.

The programmer has no control over when the finalizer is called; the garbage collector decides when to call it. The garbage collector checks for objects that are no longer being used by the application. If it considers an object eligible for finalization, it calls the finalizer (if any) and reclaims the memory used to store the object. It's possible to force garbage collection by calling Collect, but most of the time, this call should be avoided because it may create performance issues.

> **Note**
>
> Whether or not finalizers are run as part of application termination is specific to each implementation of .NET. When an application terminates, .NET Framework makes every reasonable effort to call finalizers for objects that haven't yet been garbage collected, unless such cleanup has been suppressed (by a call to the library method `GC.SuppressFinalize`, for example). .NET 5 (including .NET Core) and later

versions don't call finalizers as part of application termination. For more information, see GitHub issue dotnet/csharpstandard #291.

If you need to perform cleanup reliably when an application exits, register a handler for the System.AppDomain.ProcessExit event. That handler would ensure IDisposable.Dispose() (or, IAsyncDisposable.DisposeAsync()) has been called for all objects that require cleanup before application exit. Because you can't call *Finalize* directly, and you can't guarantee the garbage collector calls all finalizers before exit, you must use `Dispose` or `DisposeAsync` to ensure resources are freed.

Using finalizers to release resources

In general, C# does not require as much memory management on the part of the developer as languages that don't target a runtime with garbage collection. This is because the .NET garbage collector implicitly manages the allocation and release of memory for your objects. However, when your application encapsulates unmanaged resources, such as windows, files, and network connections, you should use finalizers to free those resources. When the object is eligible for finalization, the garbage collector runs the `Finalize` method of the object.

Explicit release of resources

If your application is using an expensive external resource, we also recommend that you provide a way to explicitly release the resource before the garbage collector frees the object. To release the resource, implement a `Dispose` method from the IDisposable interface that performs the necessary cleanup for the object. This can considerably improve the performance of the application. Even with this explicit control over resources, the finalizer becomes a safeguard to clean up resources if the call to the `Dispose` method fails.

For more information about cleaning up resources, see the following articles:

- Cleaning Up Unmanaged Resources
- Implementing a Dispose Method
- Implementing a DisposeAsync Method
- using statement

Example

The following example creates three classes that make a chain of inheritance. The class `First` is the base class, `Second` is derived from `First`, and `Third` is derived from `Second`. All three have finalizers. In `Main`, an instance of the most-derived class is created. The output from this code depends on which implementation of .NET the application targets:

- .NET Framework: The output shows that the finalizers for the three classes are called automatically when the application terminates, in order from the most-derived to the least-derived.
- .NET 5 (including .NET Core) or a later version: There's no output, because this implementation of .NET doesn't call finalizers when the application terminates.

C#

```
class First
{
    ~First()
    {
        System.Diagnostics.Trace.WriteLine("First's finalizer is called.");
    }
}

class Second : First
{
    ~Second()
    {
        System.Diagnostics.Trace.WriteLine("Second's finalizer is called.");
    }
}

class Third : Second
{
    ~Third()
    {
        System.Diagnostics.Trace.WriteLine("Third's finalizer is called.");
    }
}

/*
Test with code like the following:
    Third t = new Third();
    t = null;

When objects are finalized, the output would be:
Third's finalizer is called.
Second's finalizer is called.
First's finalizer is called.
*/
```

C# language specification

For more information, see the Finalizers section of the C# Language Specification.

See also

- IDisposable
- Constructors
- Garbage Collection

Object and collection initializers (C# programming guide)

Summarize this article for me

C# enables you to instantiate an object or collection and perform member assignments in a single statement.

Object initializers

Object initializers let you assign values to any accessible fields or properties of an object when you create it. You don't need to invoke a constructor and then use assignment statements. The object initializer syntax enables you to specify arguments for a constructor or omit the arguments and parentheses. For guidance on using object initializers consistently, see Use object initializers (style rule IDE0017). The following example shows how to use an object initializer with a named type, `Cat`, and how to invoke the parameterless constructor. Note the use of automatically implemented properties in the `Cat` class. For more information, see Automatically implemented properties.

C#

```
public class Cat
{
    // Automatically implemented properties.
    public int Age { get; set; }
    public string? Name { get; set; }

    public Cat()
    {
    }

    public Cat(string name)
    {
        this.Name = name;
    }
}
```

C#

```
Cat cat = new Cat { Age = 10, Name = "Fluffy" };
Cat sameCat = new Cat("Fluffy"){ Age = 10 };
```

The object initializer syntax allows you to create an instance and assign the newly created object, with its assigned properties, to the variable in the assignment.

Starting with nested object properties, you can use object initializer syntax without the `new` keyword. This syntax, `Property = { ... }`, enables you to initialize members of existing nested objects, which is useful with read-only properties. For more information, see Object Initializers with class-typed properties.

Object initializers can set indexers, in addition to assigning fields and properties. Consider this basic `Matrix` class:

C#

```
public class Matrix
{
    private double[,] storage = new double[3, 3];

    public double this[int row, int column]
    {
        // The embedded array will throw out of range exceptions as appropriate.
        get { return storage[row, column]; }
        set { storage[row, column] = value; }
    }
}
```

You can initialize the identity matrix by using the following code:

C#

```
var identity = new Matrix
{
    [0, 0] = 1.0,
    [0, 1] = 0.0,
    [0, 2] = 0.0,

    [1, 0] = 0.0,
    [1, 1] = 1.0,
    [1, 2] = 0.0,

    [2, 0] = 0.0,
    [2, 1] = 0.0,
    [2, 2] = 1.0,
};
```

Any accessible indexer that contains an accessible setter can be used as one of the expressions in an object initializer, regardless of the number or types of arguments. The index arguments form the left side of the assignment, and the value is the right side of the expression. For example, the following initializers are all valid if `IndexersExample` has the appropriate indexers:

C#

```
var thing = new IndexersExample
{
    name = "object one",
    [1] = '1',
    [2] = '4',
    [3] = '9',
    Size = Math.PI,
    ['C',4] = "Middle C"
}
```

For the preceding code to compile, the `IndexersExample` type must have the following members:

C#

```
public string name;
public double Size { set { ... }; }
public char this[int i] { set { ... }; }
public string this[char c, int i] { set { ... }; }
```

Object initializers with anonymous types

Although you can use object initializers in any context, they're especially useful in Language-Integrated Query (LINQ) expressions. Query expressions frequently use anonymous types, which you can only initialize by using an object initializer, as shown in the following declaration.

C#

```
var pet = new { Age = 10, Name = "Fluffy" };
```

By using anonymous types, the `select` clause in a LINQ query expression can transform objects of the original sequence into objects whose value and shape differ from the original. You might want to store only a part of the information from each object in a sequence. In the following example, assume that a product object (`p`) contains many fields and methods, and that you're only interested in creating a sequence of objects that contain the product name and the unit price.

C#

```
var productInfos =
    from p in products
    select new { p.ProductName, p.UnitPrice };
```

When you execute this query, the `productInfos` variable contains a sequence of objects that you can access in a `foreach` statement as shown in this example:

C#

```
foreach(var p in productInfos){...}
```

Each object in the new anonymous type has two public properties that receive the same names as the properties or fields in the original object. You can also rename a field when you create an anonymous type. The following example renames the `UnitPrice` field to `Price`.

C#

```
select new {p.ProductName, Price = p.UnitPrice};
```

Object initializers with the `required` modifier

Use the `required` keyword to force callers to set the value of a property or field by using an object initializer. You don't need to set required properties as constructor parameters. The compiler ensures all callers initialize those values.

C#

```
public class Pet
{
    public required int Age;
    public string Name;
}

// `Age` field is necessary to be initialized.
// You don't need to initialize `Name` property
var pet = new Pet() { Age = 10};

// Compiler error:
// Error CS9035 Required member 'Pet.Age' must be set in the object initializer or
attribute constructor.
// var pet = new Pet();
```

It's a typical practice to guarantee that your object is properly initialized, especially when you have multiple fields or properties to manage and don't want to include them all in the constructor.

Object initializers with the `init` accessor

By using an `init` accessor, you can make sure the object doesn't change after initializtion. It helps to restrict the setting of the property value.

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; init; }
}

// The `LastName` property can be set only during initialization. It CAN'T be
modified afterwards.
// The `FirstName` property can be modified after initialization.
var pet = new Person() { FirstName = "Joe", LastName = "Doe"};

// You can assign the FirstName property to a different value.
pet.FirstName = "Jane";

// Compiler error:
// Error CS8852  Init - only property or indexer 'Person.LastName' can only be
assigned in an object initializer,
//               or on 'this' or 'base' in an instance constructor or an 'init'
accessor.
// pet.LastName = "Kowalski";
```

Required init-only properties support immutable structures while allowing natural syntax for users of the type.

Object initializers with class-typed properties

When you initialize objects with class-typed properties, you can use two different syntaxes:

1. **Object initializer without `new` keyword**: `Property = { ... }`
2. **Object initializer with `new` keyword**: `Property = new() { ... }`

These syntaxes behave differently. The following example demonstrates both approaches:

C#

```
public class HowToClassTypedInitializer
{
    public class EmbeddedClassTypeA
    {
        public int I { get; set; }
        public bool B { get; set; }
        public string S { get; set; }
        public EmbeddedClassTypeB ClassB { get; set; }
```

```
        public override string ToString() => $"{I}|{B}|{S}|||{ClassB}";

        public EmbeddedClassTypeA()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeA constructor. Values
are: {this}");
            I = 3;
            B = true;
            S = "abc";
            ClassB = new() { BB = true, BI = 43 };
            Console.WriteLine($"Exiting EmbeddedClassTypeA constructor. Values are:
{this})");
        }
    }

    public class EmbeddedClassTypeB
    {
        public int BI { get; set; }
        public bool BB { get; set; }
        public string BS { get; set; }

        public override string ToString() => $"{BI}|{BB}|{BS}";

        public EmbeddedClassTypeB()
        {
            Console.WriteLine($"Entering EmbeddedClassTypeB constructor. Values
are: {this}");
            BI = 23;
            BB = false;
            BS = "BBBabc";
            Console.WriteLine($"Exiting EmbeddedClassTypeB constructor. Values are:
{this})");
        }
    }

    public static void Main()
    {
        var a = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = { BI = 100003 }
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA: {a}");

        var a2 = new EmbeddedClassTypeA
        {
            I = 103,
            B = false,
            ClassB = new() { BI = 100003 } //New instance
        };
        Console.WriteLine($"After initializing EmbeddedClassTypeA a2: {a2}");
    }

    // Output:
```

```
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //After initializing EmbeddedClassTypeA: 103|False|abc|||100003|True|BBBabc
    //Entering EmbeddedClassTypeA constructor Values are: 0|False||||
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //Exiting EmbeddedClassTypeA constructor Values are:
3|True|abc|||43|True|BBBabc)
    //Entering EmbeddedClassTypeB constructor Values are: 0|False|
    //Exiting EmbeddedClassTypeB constructor Values are: 23|False|BBBabc)
    //After initializing EmbeddedClassTypeA a2: 103|False|abc|||100003|False|BBBabc
}
```

Key differences

- **Without `new` keyword** (`ClassB = { BI = 100003 }`): This syntax modifies the existing instance of the property that the object constructor created. It calls member initializers on the existing object.
- **With `new` keyword** (`ClassB = new() { BI = 100003 }`): This syntax creates a new instance and assigns it to the property, replacing any existing instance.

The initializer without `new` reuses the current instance. In the previous example, ClassB's values are: `100003` (new value assigned), `true` (kept from EmbeddedClassTypeA's initialization), `BBBabc` (unchanged default from EmbeddedClassTypeB).

Object initializers without `new` for read-only properties

The syntax without `new` is useful with read-only properties. You can't assign a new instance but you can still initialize the existing instance's members:

C#

```
public class ReadOnlyPropertyExample
{
    public class Settings
    {
        public string Theme { get; set; } = "Light";
        public int FontSize { get; set; } = 12;
    }

    public class Application
    {
        public string Name { get; set; } = "";
```

```
        // This property is read-only - it can only be set during construction
        public Settings AppSettings { get; } = new();
    }

    public static void Example()
    {
        // You can still initialize the nested object's properties
        // even though AppSettings property has no setter
        var app = new Application
        {
            Name = "MyApp",
            AppSettings = { Theme = "Dark", FontSize = 14 }
        };

        // This would cause a compile error because AppSettings has no setter:
        // app.AppSettings = new Settings { Theme = "Dark", FontSize = 14 };

        Console.WriteLine($"App: {app.Name}, Theme: {app.AppSettings.Theme}, Font
Size: {app.AppSettings.FontSize}");
    }
}
```

This approach allows you to initialize nested objects even when the containing property doesn't have a setter.

Collection initializers

Collection initializers let you specify one or more element initializers when you initialize a collection type that implements IEnumerable and has an `Add` method with the appropriate signature as an instance method or an extension method. The element initializers can be a value, an expression, or an object initializer. By using a collection initializer, you don't have to specify multiple calls; the compiler adds the calls automatically. For guidance on using collection initializers consistently, see Use collection initializers (style rule IDE0028). Collection initializers are also useful in LINQ queries.

The following example shows two simple collection initializers:

C#

```
List<int> digits = new List<int> { 0, 1, 2, 3, 4, 5, 6, 7, 8, 9 };
List<int> digits2 = new List<int> { 0 + 1, 12 % 3, MakeInt() };
```

The following collection initializer uses object initializers to initialize objects of the `Cat` class defined in a previous example. The individual object initializers are enclosed in braces and separated by commas.

C#

```
List<Cat> cats =
[
    new Cat { Name = "Sylvester", Age = 8 },
    new Cat { Name = "Whiskers", Age = 2 },
    new Cat { Name = "Sasha", Age = 14 }
];
```

You can specify null as an element in a collection initializer if the collection's `Add` method allows it.

C#

```
List<Cat?> moreCats = new List<Cat?>
{
    new Cat{ Name = "Furrytail", Age=5 },
    new Cat{ Name = "Peaches", Age=4 },
    null
};
```

You can use a spread element to create one list that copies other list or lists.

C#

```
List<Cat> allCats = [.. cats, .. moreCats];
```

And include additional elements along with using a spread element.

C#

```
List<Cat> additionalCats = [.. cats, new Cat { Name = "Furrytail", Age = 5 }, ..
moreCats];
```

You can specify indexed elements if the collection supports read / write indexing.

C#

```
var numbers = new Dictionary<int, string>
{
    [7] = "seven",
    [9] = "nine",
    [13] = "thirteen"
};
```

The preceding sample generates code that calls the Item[TKey] to set the values. You can also initialize dictionaries and other associative containers by using the following syntax. Instead of

indexer syntax, with parentheses and an assignment, it uses an object with multiple values:

C#

```
var moreNumbers = new Dictionary<int, string>
{
    {19, "nineteen" },
    {23, "twenty-three" },
    {42, "forty-two" }
};
```

This initializer example calls Add(TKey, TValue) to add the three items into the dictionary. These two different ways to initialize associative collections have slightly different behavior because of the method calls the compiler generates. Both variants work with the `Dictionary` class. Other types might only support one or the other based on their public API.

Collection expression arguments

Starting in C# 15, use the `with(...)` element as the first element in a collection expression to pass arguments to the collection's constructor. By using this feature, you can specify capacity, comparers, or other constructor parameters directly within the collection expression syntax:

C#

```
public static void CollectionExpressionWithArgumentsExample()
{
    string[] values = ["one", "two", "three"];

    // Use with() to pass capacity to the List<T> constructor
    List<string> names = [with(capacity: values.Length * 2), .. values];

    Console.WriteLine($"Created List<string> with capacity: {names.Capacity}");
    Console.WriteLine($"List contains {names.Count} elements: [{string.Join(", ",
names)}]");

    // Use with() to pass a comparer to the HashSet<T> constructor
    HashSet<string> caseInsensitiveSet = [with(StringComparer.OrdinalIgnoreCase),
"Hello", "HELLO"];
    // caseInsensitiveSet contains only one element because "Hello" and "HELLO" are
equal

    Console.WriteLine($"HashSet with case-insensitive comparer contains
{caseInsensitiveSet.Count} elements: [{string.Join(", ", caseInsensitiveSet)}]");
    Console.WriteLine("Note: 'Hello' and 'HELLO' are treated as the same element
due to OrdinalIgnoreCase comparer");
}
```

For more information about collection expression arguments, including supported target types and restrictions, see Collection expression arguments.

Object initializers with collection read-only property initialization

Some classes have collection properties where the property is read-only, like the `Cats` property of `CatOwner` in the following case:

C#

```
public class CatOwner
{
    public IList<Cat> Cats { get; } = new List<Cat>();
}
```

You can't use the collection initializer syntax discussed earlier since the property can't be assigned a new list:

C#

```
CatOwner owner = new CatOwner
{
    Cats = new List<Cat>
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

However, you can add new entries to `Cats` by using the initialization syntax and omitting the list creation (`new List<Cat>`), as shown in the following example:

C#

```
CatOwner owner = new CatOwner
{
    Cats =
    {
        new Cat{ Name = "Sylvester", Age=8 },
        new Cat{ Name = "Whiskers", Age=2 },
        new Cat{ Name = "Sasha", Age=14 }
    }
};
```

The set of entries to add appears surrounded by braces. The preceding code is identical to writing:

C#

```
CatOwner owner = new ();
owner.Cats.Add(new Cat{ Name = "Sylvester", Age=8 });
owner.Cats.Add(new Cat{ Name = "Whiskers", Age=2 });
owner.Cats.Add(new Cat{ Name = "Sasha", Age=14 });
```

Examples

The following example combines the concepts of object and collection initializers.

C#

```
public class InitializationSample
{
    public class Cat
    {
        // Automatically implemented properties.
        public int Age { get; set; }
        public string? Name { get; set; }

        public Cat() { }

        public Cat(string name)
        {
            Name = name;
        }
    }

    public static void Main()
    {
        Cat cat = new Cat { Age = 10, Name = "Fluffy" };
        Cat sameCat = new Cat("Fluffy"){ Age = 10 };

        List<Cat> cats =
        [
            new Cat { Name = "Sylvester", Age = 8 },
            new Cat { Name = "Whiskers", Age = 2 },
            new Cat { Name = "Sasha", Age = 14 }
        ];

        List<Cat?> moreCats = new List<Cat?>
        {
            new Cat { Name = "Furrytail", Age = 5 },
            new Cat { Name = "Peaches", Age = 4 },
            null
        };
```

```
        List<Cat> allCats = [.. cats, new Cat { Name = "Łapka", Age = 5 }, cat, ..
moreCats];

        // Display results.
        foreach (Cat? c in allCats)
        {
            if (c != null)
            {
                System.Console.WriteLine(c.Name);
            }
            else
            {
                System.Console.WriteLine("List element has null value.");
            }
        }
    }
    // Output:
    // Sylvester
    // Whiskers
    // Sasha
    // Łapka
    // Fluffy
    // Furrytail
    // Peaches
    // List element has null value.
}
```

The following example shows an object that implements IEnumerable and contains an `Add` method with multiple parameters. It uses a collection initializer with multiple elements per item in the list that correspond to the signature of the `Add` method.

C#

```
public class FullExample
{
    class FormattedAddresses : IEnumerable<string>
    {
        private List<string> internalList = new();
        public IEnumerator<string> GetEnumerator() => internalList.GetEnumerator();

        System.Collections.IEnumerator
System.Collections.IEnumerable.GetEnumerator() => internalList.GetEnumerator();

        public void Add(string firstname, string lastname,
            string street, string city,
            string state, string zipcode) => internalList.Add($"""
            {firstname} {lastname}
            {street}
            {city}, {state} {zipcode}
            """
            );
```

```
    }

    public static void Main()
    {
        FormattedAddresses addresses = new FormattedAddresses()
        {
            {"John", "Doe", "123 Street", "Topeka", "KS", "00000" },
            {"Jane", "Smith", "456 Street", "Topeka", "KS", "00000" }
        };

        Console.WriteLine("Address Entries:");

        foreach (string addressEntry in addresses)
        {
            Console.WriteLine("\r\n" + addressEntry);
        }
    }

    /*
        * Prints:

        Address Entries:

        John Doe
        123 Street
        Topeka, KS 00000

        Jane Smith
        456 Street
        Topeka, KS 00000
        */
}
```

`Add` methods can use the `params` keyword to take a variable number of arguments, as shown in the following example. This example also demonstrates the custom implementation of an indexer to initialize a collection using indexes. Starting with C# 13, the `params` parameter isn't restricted to an array. It can be a collection type or interface.

C#

```
public class DictionaryExample
{
    class RudimentaryMultiValuedDictionary<TKey, TValue> :
IEnumerable<KeyValuePair<TKey, List<TValue>>> where TKey : notnull
    {
        private Dictionary<TKey, List<TValue>> internalDictionary = new
Dictionary<TKey, List<TValue>>();

        public IEnumerator<KeyValuePair<TKey, List<TValue>>> GetEnumerator() =>
internalDictionary.GetEnumerator();

        System.Collections.IEnumerator
```

```
System.Collections.IEnumerable.GetEnumerator() =>
internalDictionary.GetEnumerator();

        public List<TValue> this[TKey key]
        {
            get => internalDictionary[key];
            set => Add(key, value);
        }

        public void Add(TKey key, params TValue[] values) => Add(key,
(IEnumerable<TValue>)values);

        public void Add(TKey key, IEnumerable<TValue> values)
        {
            if (!internalDictionary.TryGetValue(key, out List<TValue>?
storedValues))
            {
                internalDictionary.Add(key, storedValues = new());
            }
            storedValues.AddRange(values);
        }
    }

    public static void Main()
    {
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary1
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", "Bob", "John", "Mary" },
                {"Group2", "Eric", "Emily", "Debbie", "Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary2
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                ["Group1"] = new List<string>() { "Bob", "John", "Mary" },
                ["Group2"] = new List<string>() { "Eric", "Emily", "Debbie",
"Jesse" }
            };
        RudimentaryMultiValuedDictionary<string, string>
rudimentaryMultiValuedDictionary3
            = new RudimentaryMultiValuedDictionary<string, string>()
            {
                {"Group1", new string []{ "Bob", "John", "Mary" } },
                { "Group2", new string[]{ "Eric", "Emily", "Debbie", "Jesse" } }
            };

        Console.WriteLine("Using first multi-valued dictionary created with a
collection initializer:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary1)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");
```

```
            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }

        Console.WriteLine("\r\nUsing second multi-valued dictionary created with a
collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary2)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
        Console.WriteLine("\r\nUsing third multi-valued dictionary created with a
collection initializer using indexing:");

        foreach (KeyValuePair<string, List<string>> group in
rudimentaryMultiValuedDictionary3)
        {
            Console.WriteLine($"\r\nMembers of group {group.Key}: ");

            foreach (string member in group.Value)
            {
                Console.WriteLine(member);
            }
        }
    }

    /*
        * Prints:

        Using first multi-valued dictionary created with a collection initializer:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using second multi-valued dictionary created with a collection initializer
using indexing:

        Members of group Group1:
```

```
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse

        Using third multi-valued dictionary created with a collection initializer
using indexing:

        Members of group Group1:
        Bob
        John
        Mary

        Members of group Group2:
        Eric
        Emily
        Debbie
        Jesse
        */
}
```

Last updated on 02/11/2026

How to initialize objects by using an object initializer (C# Programming Guide)

07/03/2025

You can use object initializers to initialize type objects in a declarative manner without explicitly invoking a constructor for the type.

The following examples show how to use object initializers with named objects. The compiler processes object initializers by first accessing the parameterless instance constructor and then processing the member initializations. Therefore, if the parameterless constructor is declared as `private` in the class, object initializers that require public access will fail.

You must use an object initializer if you're defining an anonymous type. For more information, see How to return subsets of element properties in a query.

Example

The following example shows how to initialize a new `StudentName` type by using object initializers. This example sets properties in the `StudentName` type:

C#

```
public class HowToObjectInitializers
{
    public static void Main()
    {
        // Declare a StudentName by using the constructor that has two parameters.
        StudentName student1 = new StudentName("Craig", "Playstead");

        // Make the same declaration by using an object initializer and sending
        // arguments for the first and last names. The parameterless constructor
is
        // invoked in processing this declaration, not the constructor that has
        // two parameters.
        StudentName student2 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead"
        };

        // Declare a StudentName by using an object initializer and sending
        // an argument for only the ID property. No corresponding constructor is
        // necessary. Only the parameterless constructor is used to process object
        // initializers.
        StudentName student3 = new StudentName
        {
```

```
            ID = 183
        };

        // Declare a StudentName by using an object initializer and sending
        // arguments for all three properties. No corresponding constructor is
        // defined in the class.
        StudentName student4 = new StudentName
        {
            FirstName = "Craig",
            LastName = "Playstead",
            ID = 116
        };

        Console.WriteLine(student1.ToString());
        Console.WriteLine(student2.ToString());
        Console.WriteLine(student3.ToString());
        Console.WriteLine(student4.ToString());
    }
    // Output:
    // Craig  0
    // Craig  0
    //   183
    // Craig  116

    public class StudentName
    {
        // This constructor has no parameters. The parameterless constructor
        // is invoked in the processing of object initializers.
        // You can test this by changing the access modifier from public to
        // private. The declarations in Main that use object initializers will
        // fail.
        public StudentName() { }

        // The following constructor has parameters for two of the three
        // properties.
        public StudentName(string first, string last)
        {
            FirstName = first;
            LastName = last;
        }

        // Properties.
        public string? FirstName { get; set; }
        public string? LastName { get; set; }
        public int ID { get; set; }

        public override string ToString() => FirstName + "  " + ID;
    }
}
```

Object initializers can be used to set indexers in an object. The following example defines a `BaseballTeam` class that uses an indexer to get and set players at different positions. The

initializer can assign players, based on the abbreviation for the position, or the number used for each position baseball scorecards:

C#

```
public class HowToIndexInitializer
{
    public class BaseballTeam
    {
        private string[] players = new string[9];
        private readonly List<string> positionAbbreviations = new List<string>
        {
            "P", "C", "1B", "2B", "3B", "SS", "LF", "CF", "RF"
        };

        public string this[int position]
        {
            // Baseball positions are 1 - 9.
            get { return players[position-1]; }
            set { players[position-1] = value; }
        }
        public string this[string position]
        {
            get { return players[positionAbbreviations.IndexOf(position)]; }
            set { players[positionAbbreviations.IndexOf(position)] = value; }
        }
    }

    public static void Main()
    {
        var team = new BaseballTeam
        {
            ["RF"] = "Mookie Betts",
            [4] = "Jose Altuve",
            ["CF"] = "Mike Trout"
        };

        Console.WriteLine(team["2B"]);
    }
}
```

The next example shows the order of execution of constructor and member initializations using constructor with and without parameter:

C#

```
public class ObjectInitializersExecutionOrder
{
    public static void Main()
    {
        new Person { FirstName = "Paisley", LastName = "Smith", City = "Dallas" };
        new Dog(2) { Name = "Mike" };
```

```
    }

    public class Dog
    {
        private int age;
        private string name;

        public Dog(int age)
        {
            Console.WriteLine("Hello from Dog's non-parameterless constructor");
            this.age = age;
        }

        public required string Name
        {
            get { return name; }

            set
            {
                Console.WriteLine("Hello from setter of Dog's required property
'Name'");
                name = value;
            }
        }
    }

    public class Person
    {
        private string firstName;
        private string lastName;
        private string city;

        public Person()
        {
            Console.WriteLine("Hello from Person's parameterless constructor");
        }

        public required string FirstName
        {
            get { return firstName; }

            set
            {
                Console.WriteLine("Hello from setter of Person's required property
'FirstName'");
                firstName = value;
            }
        }

        public string LastName
        {
            get { return lastName; }

            init
            {
```

```
                Console.WriteLine("Hello from setter of Person's init property
'LastName'");
                lastName = value;
            }
        }

        public string City
        {
            get { return city; }

            set
            {
                Console.WriteLine("Hello from setter of Person's property
'City'");
                city = value;
            }
        }
    }

    // Output:
    // Hello from Person's parameterless constructor
    // Hello from setter of Person's required property 'FirstName'
    // Hello from setter of Person's init property 'LastName'
    // Hello from setter of Person's property 'City'
    // Hello from Dog's non-parameterless constructor
    // Hello from setter of Dog's required property 'Name'
}
```

Object initializers without the `new` keyword

You can also use object initializer syntax without the `new` keyword to initialize properties of nested objects. This syntax is particularly useful with read-only properties:

C#

```
public class ObjectInitializerWithoutNew
{
    public class Address
    {
        public string Street { get; set; } = "";
        public string City { get; set; } = "";
        public string State { get; set; } = "";
    }

    public class Person
    {
        public string Name { get; set; } = "";
        public Address HomeAddress { get; set; } = new(); // Property with setter
    }

    public static void Examples()
```

```
    {
        // Example 1: Using object initializer WITHOUT 'new' keyword
        // This modifies the existing Address instance created in the constructor
        var person1 = new Person
        {
            Name = "Alice",
            HomeAddress = { Street = "123 Main St", City = "Anytown", State = "CA"
}
        };

        // Example 2: Using object initializer WITH 'new' keyword
        // This creates a completely new Address instance
        var person2 = new Person
        {
            Name = "Bob",
            HomeAddress = new Address { Street = "456 Oak Ave", City =
"Somewhere", State = "NY" }
        };

        // Both approaches work, but they behave differently:
        // - person1.HomeAddress is the same instance that was created in Person's
constructor
        // - person2.HomeAddress is a new instance, replacing the one from the
constructor

        Console.WriteLine($"Person 1: {person1.Name} at
{person1.HomeAddress.Street}, {person1.HomeAddress.City},
{person1.HomeAddress.State}");
        Console.WriteLine($"Person 2: {person2.Name} at
{person2.HomeAddress.Street}, {person2.HomeAddress.City},
{person2.HomeAddress.State}");
    }
}
```

This approach modifies the existing instance of the nested object rather than creating a new one. For more details and examples, see Object Initializers with class-typed properties.

See also

- Object and Collection Initializers

How to initialize a dictionary with a collection initializer (C# Programming Guide)

10/18/2025

A Dictionary<TKey,TValue> contains a collection of key/value pairs. Its Add method takes two parameters, one for the key and one for the value. One way to initialize a Dictionary<TKey,TValue>, or any collection whose `Add` method takes multiple parameters, is to enclose each set of parameters in braces as shown in the following example. Another option is to use an index initializer, also shown in the following example.

> **Note**
>
> The major difference between these two ways of initializing the collection is how duplicated keys are handled, for example:
>
> C#
>
> ```
> { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211 } },
> { 111, new StudentName { FirstName="Dina", LastName="Salimzianova", ID=317 }
> },
> ```
>
> Add method throws ArgumentException: `'An item with the same key has already been added. Key: 111'`, while the second part of example, the public read / write indexer method, quietly overwrites the already existing entry with the same key.

Example

In the following code example, a Dictionary<TKey,TValue> is initialized with instances of type `StudentName`. The first initialization uses the `Add` method with two arguments. The compiler generates a call to `Add` for each of the pairs of `int` keys and `StudentName` values. The second uses a public read / write indexer method of the `Dictionary` class:

C#

```
public class HowToDictionaryInitializer
{
    class StudentName
    {
        public string? FirstName { get; set; }
```

```
        public string? LastName { get; set; }
        public int ID { get; set; }
    }

    public static void Main()
    {
        var students = new Dictionary<int, StudentName>()
        {
            { 111, new StudentName { FirstName="Sachin", LastName="Karnik", ID=211
} },
            { 112, new StudentName { FirstName="Dina", LastName="Salimzianova",
ID=317 } },
            { 113, new StudentName { FirstName="Andy", LastName="Ruth", ID=198 } }
        };

        foreach(var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students[index].FirstName}
{students[index].LastName}");
        }
        Console.WriteLine();

        var students2 = new Dictionary<int, StudentName>()
        {
            [111] = new StudentName { FirstName="Sachin", LastName="Karnik",
ID=211 },
            [112] = new StudentName { FirstName="Dina", LastName="Salimzianova",
ID=317 } ,
            [113] = new StudentName { FirstName="Andy", LastName="Ruth", ID=198 }
        };

        foreach (var index in Enumerable.Range(111, 3))
        {
            Console.WriteLine($"Student {index} is {students2[index].FirstName}
{students2[index].LastName}");
        }
    }
}
```

Note the two pairs of braces in each element of the collection in the first declaration. The innermost braces enclose the object initializer for the `StudentName`, and the outermost braces enclose the initializer for the key/value pair to be added to the `students` Dictionary<TKey,TValue>. Finally, the whole collection initializer for the dictionary is enclosed in braces. In the second initialization, the left side of the assignment is the key and the right side is the value, using an object initializer for `StudentName`.

Use AI to generate test data for dictionary collections

You can use AI tools, such as GitHub Copilot, to quickly generate dictionary test data and validation scenarios in your C# projects.

Here's an example prompt you can use in Visual Studio Copilot Chat.

Copilot prompt

```
Generate data collections for tests to create a separate Dictionary<int, Student>
containing 10 valid Student records and 5 invalid records.
- Valid records should have realistic Name and Grade values.
- Invalid records should include cases such as missing Name, Grade < 0, or Grade >
100.
- This dictionary should be used only for testing purposes and not modify existing
production code.
- Generate test code that utilizes this test data for validation scenarios.
- Call test method to run the test.
```

Review Copilot's suggestions before applying them.

For more information about GitHub Copilot, see GitHub's FAQs.

See also

- Object and Collection Initializers

Nested types (C# programming guide)

A nested type is a type that you define within a class, struct, or interface. For example:

C#

```
public class Container
{
    class Nested
    {
        Nested() { }
    }
}
```

Regardless of whether the outer type is a class, interface, or struct, nested types default to private. You can access them only from their containing type. In the preceding example, external types can't access the `Nested` class.

You can also specify an access modifier to define the accessibility of a nested type, as follows:

- Nested types of a **class** can be public, protected, internal, protected internal, private, or private protected.

  However, defining a `protected`, `protected internal`, or `private protected` nested class inside a sealed class generates compiler warning CS0628, "new protected member declared in sealed class."

  Also be aware that making a nested type externally visible violates the code quality rule CA1034 "Nested types should not be visible".

- Nested types of a **struct** can be public, internal, or private.

The following example makes the `Nested` class public:

C#

```
public class Container
{
    public class Nested
    {
        Nested() { }
    }
}
```

The nested, or inner, type can access the containing, or outer, type. To access the containing type, pass it as an argument to the constructor of the nested type. For example:

C#

```
public class Container
{
    public class Nested
    {
        private Container? parent;

        public Nested()
        {
        }
        public Nested(Container parent)
        {
            this.parent = parent;
        }
    }
}
```

A nested type has access to all of the members that are accessible to its containing type. It can access private and protected members of the containing type, including any inherited protected members.

In the previous declaration, the full name of class `Nested` is `Container.Nested`. This is the name used to create a new instance of the nested class, as follows:

C#

```
Container.Nested nest = new Container.Nested();
```

See also

- The C# type system
- Access Modifiers
- Constructors
- CA1034 rule

Last updated on 12/17/2025

Partial Classes and Members (C# Programming Guide)

It's possible to split the definition of a class, a struct, an interface, or a member over two or more source files. Each source file contains a section of the type or member definition, and all parts are combined when the application is compiled.

Partial Classes

There are several situations when splitting a class definition is desirable:

- Declaring a class over separate files enables multiple programmers to work on it at the same time.
- You can add code to the class without having to recreate the source file that includes automatically generated source. Visual Studio uses this approach when it creates Windows Forms, Web service wrapper code, and so on. You can create code that uses these classes without having to modify the file created by Visual Studio.
- Source generators can generate extra functionality in a class.

To split a class definition, use the partial keyword modifier. In practice, each partial class is typically defined in a separate file, making it easier to manage and expand the class over time.

The following `Employee` example demonstrates how the class might be divided across two files: Employee_Part1.cs and Employee_Part2.cs.

C#

```
// This is in Employee_Part1.cs
public partial class Employee
{
    public void DoWork()
    {
        Console.WriteLine("Employee is working.");
    }
}

// This is in Employee_Part2.cs
public partial class Employee
{
    public void GoToLunch()
    {
        Console.WriteLine("Employee is at lunch.");
    }
}

//Main program demonstrating the Employee class usage
```

```
public class Program
{
    public static void Main()
    {
        Employee emp = new Employee();
        emp.DoWork();
        emp.GoToLunch();
    }
}

// Expected Output:
// Employee is working.
// Employee is at lunch.
```

The `partial` keyword indicates that other parts of the class, struct, or interface can be defined in the namespace. All the parts must use the `partial` keyword. All the parts must be available at compile time to form the final type. All the parts must have the same accessibility, such as `public`, `private`, and so on.

If any part is declared abstract, then the whole type is considered abstract. If any part is declared sealed, then the whole type is considered sealed. If any part declares a base type, then the whole type inherits that class.

All the parts that specify a base class must agree, but parts that omit a base class still inherit the base type. Parts can specify different base interfaces, and the final type implements all the interfaces listed by all the partial declarations. Any class, struct, or interface members declared in a partial definition are available to all the other parts. The final type is the combination of all the parts at compile time.

> ⓘ **Note**
>
> The `partial` modifier isn't available on delegate or enumeration declarations.

The following example shows that nested types can be partial, even if the type they're nested within isn't partial itself.

C#

```
class Container
{
    partial class Nested
    {
        void Test() { }
    }

    partial class Nested
    {
```

```
        void Test2() { }
    }
}
```

At compile time, attributes of partial-type definitions are merged. For example, consider the following declarations:

C#

```
[SerializableAttribute]
partial class Moon { }

[ObsoleteAttribute]
partial class Moon { }
```

They're equivalent to the following declarations:

C#

```
[SerializableAttribute]
[ObsoleteAttribute]
class Moon { }
```

The following are merged from all the partial-type definitions:

- XML comments. However, if both declarations of a partial member include comments, only the comments from the implementing member are included.
- interfaces
- generic-type parameter attributes
- class attributes
- members

For example, consider the following declarations:

C#

```
partial class Earth : Planet, IRotate { }
partial class Earth : IRevolve { }
```

They're equivalent to the following declarations:

C#

```
class Earth : Planet, IRotate, IRevolve { }
```

Restrictions

There are several rules to follow when you're working with partial class definitions:

- All partial-type definitions meant to be parts of the same type must be modified with `partial`. For example, the following class declarations generate an error:

  C#

  ```
  public partial class A { }
  //public class A { }  // Error, must also be marked partial
  ```

- The `partial` modifier can only appear immediately before the keyword `class`, `struct`, or `interface`.
- Nested partial types are allowed in partial-type definitions as illustrated in the following example:

  C#

  ```
  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }

  partial class ClassWithNestedClass
  {
      partial class NestedClass { }
  }
  ```

- All partial-type definitions meant to be parts of the same type must be defined in the same assembly and the same module (.exe or .dll file). Partial definitions can't span multiple modules.
- The class name and generic-type parameters must match on all partial-type definitions. Generic types can be partial. Each partial declaration must use the same parameter names in the same order.
- The following keywords on a partial-type definition are optional, but if present on one partial-type definition, the same must be specified on other partial definition for the same type:
  - public
  - private
  - protected
  - internal
  - abstract
  - sealed
  - base class
  - new modifier (nested parts)

- generic constraints

For more information, see Constraints on Type Parameters.

Examples

In the following example, the fields and constructor of the `Coords` class are declared in one partial class definition (`Coords_Part1.cs`), and the `PrintCoords` method is declared in another partial class definition (`Coords_Part2.cs`). This separation demonstrates how partial classes can be divided across multiple files for easier maintainability.

C#

```
// This is in Coords_Part1.cs
public partial class Coords
{
    private int x;
    private int y;

    public Coords(int x, int y)
    {
        this.x = x;
        this.y = y;
    }
}

// This is in Coords_Part2.cs
public partial class Coords
{
    public void PrintCoords()
    {
        Console.WriteLine("Coords: {0},{1}", x, y);
    }
}

// Main program demonstrating the Coords class usage
class TestCoords
{
    static void Main()
    {
        Coords myCoords = new Coords(10, 15);
        myCoords.PrintCoords();

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
// Output: Coords: 10,15
```

The following example shows that you can also develop partial structs and interfaces.

C#

```
partial interface ITest
{
    void Interface_Test();
}

partial interface ITest
{
    void Interface_Test2();
}

partial struct S1
{
    void Struct_Test() { }
}

partial struct S1
{
    void Struct_Test2() { }
}
```

Partial Members

A partial class or struct can contain a partial member. One part of the class contains the signature of the member. An implementation can be defined in the same part or another part.

An implementation isn't required for a partial method when the signature obeys the following rules:

- The declaration doesn't include any access modifiers. The method has private access by default.
- The return type is void.
- None of the parameters have the out modifier.
- The method declaration can't include any of the following modifiers:
  - virtual
  - override
  - sealed
  - new
  - extern

The method and all calls to the method are removed at compile time when there's no implementation.

Any member that doesn't conform to all those restrictions, including constructors, properties, indexers, and events, must provide an implementation. That implementation might be supplied by a *source generator*. Partial properties can't be implemented using automatically implemented properties. The compiler can't distinguish between an automatically implemented property, and the declaring declaration of a partial property.

Beginning with C# 13, the implementing declaration for a partial property can use field backed properties to define the implementing declaration. A field backed property provides a concise syntax where the `field` keyword accesses the compiler synthesized backing field for the property. For example, you could write the following code:

C#

```
// in file1.cs
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get; set; }
}

// In file2.cs
public partial class PropertyBag
{
    // Defining declaration
    public partial int MyProperty { get => field; set; }
}
```

You can use `field` in either the `get` or `set` accessor, or both.

Partial members enable the implementer of one part of a class to declare a member. The implementer of another part of the class can define that member. There are two scenarios where this separation is useful: templates that generate boilerplate code, and source generators.

- **Template code**: The template reserves a method name and signature so that generated code can call the method. These methods follow the restrictions that enable a developer to decide whether to implement the method. If the method isn't implemented, then the compiler removes the method signature and all calls to the method. The calls to the method, including any results that would occur from evaluation of arguments in the calls, have no effect at run time. Therefore, any code in the partial class can freely use a partial method, even if the implementation isn't supplied. No compile-time or run-time errors result if the method is called but not implemented.
- **Source generators**: Source generators provide an implementation for members. The human developer can add the member declaration (often with attributes read by the source generator). The developer can write code that calls these members. The source

generator runs during compilation and provides the implementation. In this scenario, the restrictions for partial members that might not be implemented often aren't followed.

C#

```
// Definition in file1.cs
partial void OnNameChanged();

// Implementation in file2.cs
partial void OnNameChanged()
{
  // method body
}
```

- Partial member declarations must begin with the contextual keyword partial.
- Partial member signatures in both parts of the partial type must match.
- Partial member can have static and unsafe modifiers.
- Partial member can be generic. Constraints must be the same on the defining and implementing member declaration. Parameter and type parameter names don't have to be the same in the implementing declaration as in the defining one.
- You can make a delegate to a partial method defined and implemented, but not to a partial method that doesn't have an implementation.

C# Language Specification

For more information, see Partial types and Partial methods in the C# Language Specification. The language specification is the definitive source for C# syntax and usage. The new features for partial members are defined in the feature specifications for extending partial methods, partial properties and indexers, and partial events and constructors.

See also

- Classes
- Structure types
- Interfaces
- partial (Type)

Last updated on 11/19/2025

How to return subsets of element properties in a query (C# Programming Guide)

Article • 03/12/2024

Use an anonymous type in a query expression when both of these conditions apply:

- You want to return only some of the properties of each source element.
- You do not have to store the query results outside the scope of the method in which the query is executed.

If you only want to return one property or field from each source element, then you can just use the dot operator in the `select` clause. For example, to return only the `ID` of each `student`, write the `select` clause as follows:

C#

```
select student.ID;
```

Example

The following example shows how to use an anonymous type to return only a subset of the properties of each source element that matches the specified condition.

C#

```
private static void QueryByScore()
{
    // Create the query. var is required because
    // the query produces a sequence of anonymous types.
    var queryHighScores =
        from student in students
        where student.ExamScores[0] > 95
        select new { student.FirstName, student.LastName };

    // Execute the query.
    foreach (var obj in queryHighScores)
    {
        // The anonymous type's properties were not named. Therefore
        // they have the same names as the Student properties.
        Console.WriteLine(obj.FirstName + ", " + obj.LastName);
    }
}
```

```
/* Output:
Adams, Terry
Fakhouri, Fadi
Garcia, Cesar
Omelchenko, Svetlana
Zabokritski, Eugene
*/
```

Note that the anonymous type uses the source element's names for its properties if no names are specified. To give new names to the properties in the anonymous type, write the `select` statement as follows:

C#

```
select new { First = student.FirstName, Last = student.LastName };
```

If you try this in the previous example, then the `Console.WriteLine` statement must also change:

C#

```
Console.WriteLine(student.First + " " + student.Last);
```

Compiling the Code

To run this code, copy and paste the class into a C# console application with a `using` directive for System.Linq.

See also

- Anonymous Types
- LINQ in C#

Explicit Interface Implementation (C# Programming Guide)

Article • 09/29/2022

If a class implements two interfaces that contain a member with the same signature, then implementing that member on the class will cause both interfaces to use that member as their implementation. In the following example, all the calls to `Paint` invoke the same method. This first sample defines the types:

C#

```
public interface IControl
{
    void Paint();
}
public interface ISurface
{
    void Paint();
}
public class SampleClass : IControl, ISurface
{
    // Both ISurface.Paint and IControl.Paint call this method.
    public void Paint()
    {
        Console.WriteLine("Paint method in SampleClass");
    }
}
```

The following sample calls the methods:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
sample.Paint();
control.Paint();
surface.Paint();

// Output:
// Paint method in SampleClass
// Paint method in SampleClass
// Paint method in SampleClass
```

But you might not want the same implementation to be called for both interfaces. To call a different implementation depending on which interface is in use, you can implement an interface member explicitly. An explicit interface implementation is a class member that is only called through the specified interface. Name the class member by prefixing it with the name of the interface and a period. For example:

C#

```
public class SampleClass : IControl, ISurface
{
    void IControl.Paint()
    {
        System.Console.WriteLine("IControl.Paint");
    }
    void ISurface.Paint()
    {
        System.Console.WriteLine("ISurface.Paint");
    }
}
```

The class member `IControl.Paint` is only available through the `IControl` interface, and `ISurface.Paint` is only available through `ISurface`. Both method implementations are separate, and neither are available directly on the class. For example:

C#

```
SampleClass sample = new SampleClass();
IControl control = sample;
ISurface surface = sample;

// The following lines all call the same method.
//sample.Paint(); // Compiler error.
control.Paint();  // Calls IControl.Paint on SampleClass.
surface.Paint();  // Calls ISurface.Paint on SampleClass.

// Output:
// IControl.Paint
// ISurface.Paint
```

Explicit implementation is also used to resolve cases where two interfaces each declare different members of the same name such as a property and a method. To implement both interfaces, a class has to use explicit implementation either for the property `P`, or the method `P`, or both, to avoid a compiler error. For example:

C#

```
interface ILeft
{
    int P { get;}
}
interface IRight
{
    int P();
}

class Middle : ILeft, IRight
{
    public int P() { return 0; }
    int ILeft.P { get { return 0; } }
}
```

An explicit interface implementation doesn't have an access modifier since it isn't accessible as a member of the type it's defined in. Instead, it's only accessible when called through an instance of the interface. If you specify an access modifier for an explicit interface implementation, you get compiler error CS0106. For more information, see interface (C# Reference).

You can define an implementation for members declared in an interface. If a class inherits a method implementation from an interface, that method is only accessible through a reference of the interface type. The inherited member doesn't appear as part of the public interface. The following sample defines a default implementation for an interface method:

C#

```
public interface IControl
{
    void Paint() => Console.WriteLine("Default Paint method");
}
public class SampleClass : IControl
{
    // Paint() is inherited from IControl.
}
```

The following sample invokes the default implementation:

C#

```
var sample = new SampleClass();
//sample.Paint();// "Paint" isn't accessible.
var control = sample as IControl;
control.Paint();
```

Any class that implements the `IControl` interface can override the default `Paint` method, either as a public method, or as an explicit interface implementation.

See also

- Object oriented programming
- Interfaces
- Inheritance

How to explicitly implement interface members (C# Programming Guide)

Article • 03/12/2024

This example declares an interface, `IDimensions`, and a class, `Box`, which explicitly implements the interface members `GetLength` and `GetWidth`. The members are accessed through the interface instance `dimensions`.

Example

C#

```
interface IDimensions
{
    float GetLength();
    float GetWidth();
}

class Box : IDimensions
{
    float _lengthInches;
    float _widthInches;

    Box(float length, float width)
    {
        _lengthInches = length;
        _widthInches = width;
    }
    // Explicit interface member implementation:
    float IDimensions.GetLength()
    {
        return _lengthInches;
    }
    // Explicit interface member implementation:
    float IDimensions.GetWidth()
    {
        return _widthInches;
    }

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new(30.0f, 20.0f);

        // Declare an interface instance dimensions:
        IDimensions dimensions = box1;

        // The following commented lines would produce compilation
```

```
        // errors because they try to access an explicitly implemented
        // interface member from a class instance:
        //Console.WriteLine($"Length: {box1.GetLength()}");
        //Console.WriteLine($"Width: {box1.GetWidth()}");

        // Print out the dimensions of the box by calling the methods
        // from an instance of the interface:
        Console.WriteLine($"Length: {dimensions.GetLength()}");
        Console.WriteLine($"Width: {dimensions.GetWidth()}");
    }
}
/* Output:
    Length: 30
    Width: 20
*/
```

Robust Programming

- Notice that the following lines, in the `Main` method, are commented out because they would produce compilation errors. An interface member that is explicitly implemented cannot be accessed from a class instance:

  C#

  ```
  //Console.WriteLine($"Length: {box1.GetLength()}");
  //Console.WriteLine($"Width: {box1.GetWidth()}");
  ```

- Notice also that the following lines, in the `Main` method, successfully print out the dimensions of the box because the methods are being called from an instance of the interface:

  C#

  ```
  Console.WriteLine($"Length: {dimensions.GetLength()}");
  Console.WriteLine($"Width: {dimensions.GetWidth()}");
  ```

See also

- Object oriented programming
- Interfaces
- How to explicitly implement members of two interfaces

How to explicitly implement members of two interfaces (C# Programming Guide)

Article • 03/12/2024

Explicit interface implementation also allows the programmer to implement two interfaces that have the same member names and give each interface member a separate implementation. This example displays the dimensions of a box in both metric and English units. The Box class implements two interfaces IEnglishDimensions and IMetricDimensions, which represent the different measurement systems. Both interfaces have identical member names, Length and Width.

Example

C#

```
// Declare the English units interface:
interface IEnglishDimensions
{
    float Length();
    float Width();
}

// Declare the metric units interface:
interface IMetricDimensions
{
    float Length();
    float Width();
}

// Declare the Box class that implements the two interfaces:
// IEnglishDimensions and IMetricDimensions:
class Box : IEnglishDimensions, IMetricDimensions
{
    float _lengthInches;
    float _widthInches;

    public Box(float lengthInches, float widthInches)
    {
        _lengthInches = lengthInches;
        _widthInches = widthInches;
    }

    // Explicitly implement the members of IEnglishDimensions:
    float IEnglishDimensions.Length() => _lengthInches;
```

```
    float IEnglishDimensions.Width() => _widthInches;

    // Explicitly implement the members of IMetricDimensions:
    float IMetricDimensions.Length() => _lengthInches * 2.54f;

    float IMetricDimensions.Width() => _widthInches * 2.54f;

    static void Main()
    {
        // Declare a class instance box1:
        Box box1 = new(30.0f, 20.0f);

        // Declare an instance of the English units interface:
        IEnglishDimensions eDimensions = box1;

        // Declare an instance of the metric units interface:
        IMetricDimensions mDimensions = box1;

        // Print dimensions in English units:
        Console.WriteLine($"Length(in): {eDimensions.Length()}");
        Console.WriteLine($"Width (in): {eDimensions.Width()}");

        // Print dimensions in metric units:
        Console.WriteLine($"Length(cm): {mDimensions.Length()}");
        Console.WriteLine($"Width (cm): {mDimensions.Width()}");
    }
}
/* Output:
    Length(in): 30
    Width (in): 20
    Length(cm): 76.2
    Width (cm): 50.8
*/
```

Robust Programming

If you want to make the default measurements in English units, implement the methods Length and Width normally, and explicitly implement the Length and Width methods from the IMetricDimensions interface:

C#

```
// Normal implementation:
public float Length() => _lengthInches;
public float Width() => _widthInches;

// Explicit implementation:
float IMetricDimensions.Length() => _lengthInches * 2.54f;
float IMetricDimensions.Width() => _widthInches * 2.54f;
```

In this case, you can access the English units from the class instance and access the metric units from the interface instance:

```C#
public static void Test()
{
    Box box1 = new(30.0f, 20.0f);
    IMetricDimensions mDimensions = box1;

    Console.WriteLine($"Length(in): {box1.Length()}");
    Console.WriteLine($"Width (in): {box1.Width()}");
    Console.WriteLine($"Length(cm): {mDimensions.Length()}");
    Console.WriteLine($"Width (cm): {mDimensions.Width()}");
}
```

See also

- Object oriented programming
- Interfaces
- How to explicitly implement interface members

Delegates (C# Programming Guide)

Article • 03/13/2025

A delegate is a type that represents references to methods with a particular parameter list and return type. When you instantiate a delegate, you can associate the delegate instance with any method that has a compatible signature and return type. You can invoke (or call) the method through the delegate instance.

Delegates are used to pass methods as arguments to other methods. Event handlers are essentially methods you invoke through delegates. When you create a custom method, a class such as a Windows control can call your method when a certain event occurs.

The following example shows a delegate declaration:

```
C#

public delegate int PerformCalculation(int x, int y);
```

You can assign any method from any accessible class or struct that matches the delegate type to the delegate. The method can be either static or an instance method. The flexibility allows you to programmatically change method calls, or plug new code into existing classes.

> **Note**
>
> In the context of method overloading, the signature of a method doesn't include the return value. However, in the context of delegates, the signature does include the return value. In other words, a method must have a compatible return type as the return type declared by the delegate.

The ability to refer to a method as a parameter makes delegates ideal for defining callback methods. You can write a method that compares two objects in your application. The method can then be used in a delegate for a sort algorithm. Because the comparison code is separate from the library, the sort method can be more general.

Function pointers support similar scenarios, where you need more control over the calling convention. The code associated with a delegate is invoked by using a virtual method added to a delegate type. When you work with function pointers, you can specify different conventions.

Explore delegate characteristics

Delegates have the following characteristics:

- Delegates allow methods to be passed as parameters.
- Delegates can be used to define callback methods.
- Delegates can be chained together, such as calling multiple methods on a single event.
- Methods don't have to match the delegate type exactly. For more information, see Using Variance in Delegates.
- Lambda expressions are a more concise way of writing inline code blocks. Lambda expressions (in certain contexts) are compiled to delegate types. For more information about lambda expressions, see Lambda expressions.

Review related articles

For more information about delegates, see the following articles:

- Using delegates
- Delegates with named versus anonymous methods
- Using variance in delegates
- How to combine delegates (multicast delegates)
- How to declare, instantiate, and use a delegate

Access the C# language specification

The language specification is the definitive source for C# syntax and usage. For more information, see Delegates in the C# Language Specification.

Related links

- Delegate
- Events

Using Delegates (C# Programming Guide)

Article • 12/22/2024

A delegate is a type that safely encapsulates a method, similar to a function pointer in C and C++. Unlike C function pointers, delegates are object-oriented, type safe, and secure. The following example declares a delegate named `Callback` that can encapsulate a method that takes a string as an argument and returns void:

C#

```
public delegate void Callback(string message);
```

A delegate object is normally constructed by providing the name of the method the delegate wraps, or with a lambda expression. A delegate can be invoked once instantiated in this manner. Invoking a delegate calls the method attached to the delegate instance. The parameters passed to the delegate by the caller are passed to the method. The delegate returns the return value, if any, from the method. For example:

C#

```
// Create a method for a delegate.
public static void DelegateMethod(string message)
{
    Console.WriteLine(message);
}
```

C#

```
// Instantiate the delegate.
Callback handler = DelegateMethod;

// Call the delegate.
handler("Hello World");
```

Delegate types are derived from the Delegate class in .NET. Delegate types are sealed, they can't be derived from, and it isn't possible to derive custom classes from Delegate. Because the instantiated delegate is an object, it can be passed as an argument, or assigned to a property. A method can accept a delegate as a parameter, and call the delegate at some later time. This is known as an asynchronous callback, and is a common method of notifying a caller when a long process completes. When a delegate is used in this fashion, the code using the delegate doesn't need any knowledge of the

implementation of the method being used. The functionality is similar to the encapsulation interfaces provide.

Another common use of callbacks is defining a custom comparison method and passing that delegate to a sort method. It allows the caller's code to become part of the sort algorithm. The following example method uses the `Del` type as a parameter:

C#

```
public static void MethodWithCallback(int param1, int param2, Callback
callback)
{
    callback("The number is: " + (param1 + param2).ToString());
}
```

You can then pass the delegate created in the preceding example to that method:

C#

```
MethodWithCallback(1, 2, handler);
```

And receive the following output to the console:

Console

```
The number is: 3
```

`MethodWithCallback` doesn't need to call the console directly—it doesn't have to be designed with a console in mind. What `MethodWithCallback` does is prepare a string and pass the string to another method. A delegated method can use any number of parameters.

When a delegate is constructed to wrap an instance method, the delegate references both the instance and the method. A delegate has no knowledge of the instance type aside from the method it wraps. A delegate can refer to any type of object as long as there's a method on that object that matches the delegate signature. When a delegate is constructed to wrap a static method, it only references the method. Consider the following declarations:

C#

```
public class MethodClass
{
    public void Method1(string message) { }
```

```
    public void Method2(string message) { }
}
```

Along with the static `DelegateMethod` shown previously, we now have three methods that you can wrap in a `Del` instance.

A delegate can call more than one method when invoked, referred to as multicasting. To add an extra method to the delegate's list of methods—the invocation list—simply requires adding two delegates using the addition or addition assignment operators ('+' or '+='). For example:

C#

```
var obj = new MethodClass();
Callback d1 = obj.Method1;
Callback d2 = obj.Method2;
Callback d3 = DelegateMethod;

//Both types of assignment are valid.
Callback allMethodsDelegate = d1 + d2;
allMethodsDelegate += d3;
```

The `allMethodsDelegate` contains three methods in its invocation list—`Method1`, `Method2`, and `DelegateMethod`. The original three delegates, `d1`, `d2`, and `d3`, remain unchanged. When `allMethodsDelegate` is invoked, all three methods are called in order. If the delegate uses reference parameters, the reference is passed sequentially to each of the three methods in turn, and any changes by one method are visible to the next method. When any of the methods throws an exception that isn't caught within the method, that exception is passed to the caller of the delegate. No subsequent methods in the invocation list are called. If the delegate has a return value and/or out parameters, it returns the return value and parameters of the last method invoked. To remove a method from the invocation list, use the subtraction or subtraction assignment operators (`-` or `-=`). For example:

C#

```
//remove Method1
allMethodsDelegate -= d1;

// copy AllMethodsDelegate while removing d2
Callback oneMethodDelegate = (allMethodsDelegate - d2)!;
```

Because delegate types are derived from `System.Delegate`, the methods and properties defined by that class can be called on the delegate. For example, to find the number of

methods in a delegate's invocation list, you can write:

```
C#
```

```
int invocationCount = d1.GetInvocationList().GetLength(0);
```

Delegates with more than one method in their invocation list derive from MulticastDelegate, which is a subclass of `System.Delegate`. The preceding code works in either case because both classes support `GetInvocationList`.

Multicast delegates are used extensively in event handling. Event source objects send event notifications to recipient objects that registered to receive that event. To register for an event, the recipient creates a method designed to handle the event, then creates a delegate for that method and passes the delegate to the event source. The source calls the delegate when the event occurs. The delegate then calls the event handling method on the recipient, delivering the event data. The event source defines the delegate type for a given event. For more, see Events.

Comparing delegates of two different types assigned at compile-time results in a compilation error. If the delegate instances are statically of the type `System.Delegate`, then the comparison is allowed, but returns false at run time. For example:

```
C#
```

```
delegate void Callback1();
delegate void Callback2();

static void method(Callback1 d, Callback2 e, System.Delegate f)
{
    // Compile-time error.
    Console.WriteLine(d == e);

    // OK at compile-time. False if the run-time type of f
    // is not the same as that of d.
    Console.WriteLine(d == f);
}
```

See also

- Delegates
- Using Variance in Delegates
- Variance in Delegates
- Using Variance for Func and Action Generic Delegates
- Events

Delegates with Named vs. Anonymous Methods (C# Programming Guide)

Article • 12/22/2024

A delegate can be associated with a named method. When you instantiate a delegate by using a named method, the method is passed as a parameter, for example:

C#

```
// Declare a delegate.
delegate void WorkCallback(int x);

// Define a named method.
void DoWork(int k) { /* ... */ }

// Instantiate the delegate using the method as a parameter.
WorkCallback d = obj.DoWork;
```

The preceding example uses a named method. Delegates constructed with a named method can encapsulate either a static method or an instance method. Named methods are the only way to instantiate a delegate in earlier versions of C#. C# enables you to instantiate a delegate and immediately specify a code block that the delegate processes when called. The block can contain either a lambda expression or an anonymous method, as shown in the following example:

C#

```
// Declare a delegate.
delegate void WorkCallback(int x);

// Instantiate the delegate using an anonymous method.
WorkCallback d = (int k) => { /* ... */ };
```

The method that you pass as a delegate parameter must have the same signature as the delegate declaration. A delegate instance can encapsulate either static or instance method.

> **Note**
>
> Although the delegate can use an **out** parameter, we do not recommend its use with multicast event delegates because you cannot know which delegate will be called.

Method groups with a single overload have a *natural type*. The compiler can infer the return type and parameter types for the delegate type:

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

Examples

The following example is a simple example of declaring and using a delegate. Notice that both the delegate, `MultiplyCallback`, and the associated method, `MultiplyNumbers`, have the same signature

C#

```
// Declare a delegate
delegate void MultiplyCallback(int i, double j);

class MathClass
{
    static void Main()
    {
        MathClass m = new MathClass();

        // Delegate instantiation using "MultiplyNumbers"
        MultiplyCallback d = m.MultiplyNumbers;

        // Invoke the delegate object.
        Console.WriteLine("Invoking the delegate using 'MultiplyNumbers':");
        for (int i = 1; i <= 5; i++)
        {
            d(i, 2);
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }

    // Declare the associated method.
    void MultiplyNumbers(int m, double n)
    {
        Console.Write(m * n + " ");
    }
}
/* Output:
    Invoking the delegate using 'MultiplyNumbers':
```

```
    2 4 6 8 10
*/
```

In the following example, one delegate is mapped to both static and instance methods and returns specific information from each.

C#

```
// Declare a delegate
delegate void Callback();

class SampleClass
{
    public void InstanceMethod()
    {
        Console.WriteLine("A message from the instance method.");
    }

    static public void StaticMethod()
    {
        Console.WriteLine("A message from the static method.");
    }
}

class TestSampleClass
{
    static void Main()
    {
        var sc = new SampleClass();

        // Map the delegate to the instance method:
        Callback d = sc.InstanceMethod;
        d();

        // Map to the static method:
        d = SampleClass.StaticMethod;
        d();
    }
}
/* Output:
    A message from the instance method.
    A message from the static method.
*/
```

See also

- Delegates
- How to combine delegates (Multicast Delegates)
- Events

How to combine delegates (Multicast Delegates) (C# Programming Guide)

Summarize this article for me

This example demonstrates how to create multicast delegates. A useful property of delegate objects is that you can assign multiple methods to one delegate instance by using the `+` operator. The multicast delegate contains a list of the assigned delegates. When you call the multicast delegate, it invokes the delegates in the list, in order. You can only combine delegates of the same type. You can use the `-` operator to remove a component delegate from a multicast delegate.

C#

```
using System;


namespace DelegateExamples;

// Define a custom delegate that has a string parameter and returns void.
delegate void CustomCallback(string s);

class TestClass
{
    // Define two methods that have the same signature as CustomCallback.
    static void Hello(string s)
    {
        Console.WriteLine($"  Hello, {s}!");
    }

    static void Goodbye(string s)
    {
        Console.WriteLine($"  Goodbye, {s}!");
    }

    static void Main()
    {
        // Declare instances of the custom delegate.
        CustomCallback hiDel, byeDel, multiDel, multiMinusHiDel;

        // In this example, you can omit the custom delegate if you
        // want to and use Action<string> instead.
        //Action<string> hiDel, byeDel, multiDel, multiMinusHiDel;

        // Initialize the delegate object hiDel that references the
        // method Hello.
        hiDel = Hello;

        // Initialize the delegate object byeDel that references the
        // method Goodbye.
```

```
        byeDel = Goodbye;

        // The two delegates, hiDel and byeDel, are combined to
        // form multiDel.
        multiDel = hiDel + byeDel;

        // Remove hiDel from the multicast delegate, leaving byeDel,
        // which calls only the method Goodbye.
        multiMinusHiDel = (multiDel - hiDel)!;

        Console.WriteLine("Invoking delegate hiDel:");
        hiDel("A");
        Console.WriteLine("Invoking delegate byeDel:");
        byeDel("B");
        Console.WriteLine("Invoking delegate multiDel:");
        multiDel("C");
        Console.WriteLine("Invoking delegate multiMinusHiDel:");
        multiMinusHiDel("D");
    }
}
/* Output:
Invoking delegate hiDel:
  Hello, A!
Invoking delegate byeDel:
  Goodbye, B!
Invoking delegate multiDel:
  Hello, C!
  Goodbye, C!
Invoking delegate multiMinusHiDel:
  Goodbye, D!
*/
```

> **Note**
>
> You can add the same delegate to a multicast delegate multiple times. When you call the multicast delegate, it invokes all the delegates in the list, including duplicates. When you remove a delegate from a multicast delegate, it removes the rightmost matching entry, so only one instance is removed if there are multiple copies.

See also

- MulticastDelegate
- Events

Last updated on 03/06/2026

How to declare, instantiate, and use a Delegate (C# Programming Guide)

Article • 12/22/2024

You can declare delegates using any of the following methods:

- Declare a delegate type and declare a method with a matching signature:

```
C#
```

```
// Declare a delegate.
delegate void NotifyCallback(string str);

// Declare a method with the same signature as the delegate.
static void Notify(string name)
{
    Console.WriteLine($"Notification received for: {name}");
}
```

```
C#
```

```
// Create an instance of the delegate.
NotifyCallback del1 = new NotifyCallback(Notify);
```

- Assign a method group to a delegate type:

```
C#
```

```
NotifyCallback del2 = Notify;
```

- Declare an anonymous method

```
C#
```

```
// Instantiate NotifyCallback by using an anonymous method.
NotifyCallback del3 = delegate (string name)
{
    Console.WriteLine($"Notification received for: {name}");
};
```

- Use a lambda expression:

```
C#
```

```
// Instantiate NotifyCallback by using a lambda expression.
NotifyCallback del4 = name => Console.WriteLine($"Notification received
for: {name}");
```

For more information, see Lambda Expressions.

The following example illustrates declaring, instantiating, and using a delegate. The `BookDB` class encapsulates a bookstore database that maintains a database of books. It exposes a method, `ProcessPaperbackBooks`, which finds all paperback books in the database and calls a delegate for each one. The `delegate` type is named `ProcessBookCallback`. The `Test` class uses this class to print the titles and average price of the paperback books.

The use of delegates promotes good separation of functionality between the bookstore database and the client code. The client code has no knowledge of how the books are stored or how the bookstore code finds paperback books. The bookstore code has no knowledge of what processing is performed on the paperback books after it finds them.

C#

```
using System;
using System.Collections.Generic;

// A set of classes for handling a bookstore:
namespace Bookstore;

// Describes a book in the book list:
public record struct Book(string Title, string Author, decimal Price, bool
Paperback);

// Declare a delegate type for processing a book:
public delegate void ProcessBookCallback(Book book);

// Maintains a book database.
public class BookDB
{
    // List of all books in the database:
    List<Book> list = new();

    // Add a book to the database:
    public void AddBook(string title, string author, decimal price, bool
paperBack) =>
        list.Add(new Book(title, author, price, paperBack));

    // Call a passed-in delegate on each paperback book to process it:
    public void ProcessPaperbackBooks(ProcessBookCallback processBook)
    {
        foreach (Book b in list)
        {
```

```
            if (b.Paperback)
            {
                // Calling the delegate:
                processBook(b);
            }
        }
    }
}

// Using the Bookstore classes:

// Class to total and average prices of books:
class PriceTotaller
{
    private int countBooks = 0;
    private decimal priceBooks = 0.0m;

    internal void AddBookToTotal(Book book)
    {
        countBooks += 1;
        priceBooks += book.Price;
    }

    internal decimal AveragePrice() => priceBooks / countBooks;
}

// Class to test the book database:
class Test
{
    // Print the title of the book.
    static void PrintTitle(Book b) => Console.WriteLine($"   {b.Title}");

    // Execution starts here.
    static void Main()
    {
        BookDB bookDB = new BookDB();

        // Initialize the database with some books:
        AddBooks(bookDB);

        // Print all the titles of paperbacks:
        Console.WriteLine("Paperback Book Titles:");

        // Create a new delegate object associated with the static
        // method Test.PrintTitle:
        bookDB.ProcessPaperbackBooks(PrintTitle);

        // Get the average price of a paperback by using
        // a PriceTotaller object:
        PriceTotaller totaller = new PriceTotaller();

        // Create a new delegate object associated with the nonstatic
        // method AddBookToTotal on the object totaller:
        bookDB.ProcessPaperbackBooks(totaller.AddBookToTotal);
```

```
        Console.WriteLine($"Average Paperback Book Price:
${totaller.AveragePrice():#.##}");
    }

    // Initialize the book database with some test books:
    static void AddBooks(BookDB bookDB)
    {
        bookDB.AddBook("The C Programming Language", "Brian W. Kernighan and
Dennis M. Ritchie", 19.95m, true);
        bookDB.AddBook("The Unicode Standard 2.0", "The Unicode Consortium",
39.95m, true);
        bookDB.AddBook("The MS-DOS Encyclopedia", "Ray Duncan", 129.95m,
false);
        bookDB.AddBook("Dogbert's Clues for the Clueless", "Scott Adams",
12.00m, true);
    }
}
/* Output:
Paperback Book Titles:
   The C Programming Language
   The Unicode Standard 2.0
   Dogbert's Clues for the Clueless
Average Paperback Book Price: $23.97
*/
```

See also

- Events
- Delegates

Strings and string literals

A string is an object of type String whose value is text. Internally, the text is stored as a sequential read-only collection of Char objects. The Length property of a string represents the number of `Char` objects it contains, not the number of Unicode characters. To access the individual Unicode code points in a string, use the StringInfo object.

string vs. System.String

In C#, the `string` keyword is an alias for String; therefore, `String` and `string` are equivalent. Use the provided alias `string` as it works even without `using System;`. The `String` class provides many methods for safely creating, manipulating, and comparing strings. In addition, the C# language overloads some operators to simplify common string operations. For more information about the keyword, see string. For more information about the type and its methods, see String.

Declaring and initializing strings

You can declare and initialize strings in various ways, as shown in the following example:

C#

```
// Declare without initializing.
string message1;

// Initialize to null.
string? message2 = null;

// Initialize as an empty string.
// Use the Empty constant instead of the literal "".
string message3 = System.String.Empty;

// Initialize with a regular string literal.
string oldPath = "c:\\Program Files\\Microsoft Visual Studio 8.0";

// Initialize with a verbatim string literal.
string newPath = @"c:\Program Files\Microsoft Visual Studio 9.0";

// Use System.String if you prefer.
System.String greeting = "Hello World!";

// In local variables (i.e. within a method body)
// you can use implicit typing.
var temp = "I'm still a strongly-typed System.String!";

// Use a const string to prevent 'message4' from
// being used to store another string value.
```

```
const string message4 = "You can't get rid of me!";

// Use the String constructor only when creating
// a string from a char*, char[], or sbyte*. See
// System.String documentation for details.
char[] letters = { 'A', 'B', 'C' };
string alphabet = new string(letters);
```

You don't use the new operator to create a string object except when initializing the string with an array of chars.

Initialize a string with the Empty constant value to create a new String object whose string is of zero length. The string literal representation of a zero-length string is `""`. By initializing strings with the Empty value instead of null, you can reduce the chances of a NullReferenceException occurring. Use the static IsNullOrEmpty(String) method to verify the value of a string before you try to access it.

Immutability of strings

String objects are *immutable*: they can't be changed after they're created. All of the String methods and C# operators that appear to modify a string actually return the results in a new string object. In the following example, when the contents of `s1` and `s2` are concatenated to form a single string, the two original strings are unmodified. The `+=` operator creates a new string that contains the combined contents. That new object is assigned to the variable `s1`, and the original object assigned to `s1` is released for garbage collection because no other variable holds a reference to it.

C#

```
string s1 = "A string is more ";
string s2 = "than the sum of its chars.";

// Concatenate s1 and s2. This actually creates a new
// string object and stores it in s1, releasing the
// reference to the original object.
s1 += s2;

System.Console.WriteLine(s1);
// Output: A string is more than the sum of its chars.
```

Because a string "modification" is actually a new string creation, you must use caution when you create references to strings. If you create a reference to a string, and then "modify" the original string, the reference continues to point to the original object. The original object doesn't reflect the new object that was created when the string was modified. The following code illustrates this behavior:

C#

```
string str1 = "Hello ";
string str2 = str1;
str1 += "World";

System.Console.WriteLine(str2);
//Output: Hello
```

For more information about how to create new strings that are based on modifications such as search and replace operations on the original string, see How to modify string contents.

Quoted string literals

Quoted string literals start and end with a single double quote character (`"`) on the same line. Quoted string literals are best suited for strings that fit on a single line and don't include any escape sequences. A quoted string literal must embed escape characters, as shown in the following example:

C#

```
string columns = "Column 1\tColumn 2\tColumn 3";
//Output: Column 1        Column 2        Column 3

string rows = "Row 1\r\nRow 2\r\nRow 3";
/* Output:
    Row 1
    Row 2
    Row 3
*/

string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

Verbatim string literals

Verbatim string literals are more convenient for multi-line strings, strings that contain backslash characters, or embedded double quotes. Verbatim strings preserve new line characters as part of the string text. Use double quotation marks to embed a quotation mark inside a verbatim string. The following example shows some common uses for verbatim strings:

C#

```
string title = "\"The \u00C6olean Harp\", by Samuel Taylor Coleridge";
//Output: "The Æolean Harp", by Samuel Taylor Coleridge
```

```
string filePath = @"C:\Users\scoleridge\Documents\";
//Output: C:\Users\scoleridge\Documents\

string text = @"My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...";
/* Output:
My pensive SARA ! thy soft cheek reclined
    Thus on mine arm, most soothing sweet it is
    To sit beside our Cot,...
*/

string quote = @"Her name was ""Sara.""";
//Output: Her name was "Sara."
```

Raw string literals

You can use *raw string literals* to more easily create strings that are multi-line, or use any characters requiring escape sequences. *Raw string literals* remove the need to ever use escape sequences. You can write the string, including whitespace formatting, how you want it to appear in output. A *raw string literal*:

- Starts and ends with a sequence of at least three double quote characters (`"""`). You can use more than three consecutive characters to start and end the sequence to support string literals that contain three (or more) repeated quote characters.
- Single line raw string literals require the opening and closing quote characters on the same line.
- Multi-line raw string literals require both opening and closing quote characters on their own line.
- In multi-line raw string literals, any whitespace to the left of the closing quotes is removed from all lines of the raw string literal.
- In multi-line raw string literals, whitespace following the opening quote on the same line is ignored.
- In multi-line raw string literals, whitespace only lines following the opening quote are included in the string literal.

The following examples demonstrate these rules:

C#

```
string singleLine = """Friends say "hello" as they pass by.""";
string multiLine = """
    "Hello World!" is typically the first program someone writes.
    """;
string embeddedXML = """
       <element attr = "content">
```

```
            <body style="normal">
                Here is the main text
            </body>
            <footer>
                Excerpts from "An amazing story"
            </footer>
        </element >
        """;
// The line "<element attr = "content">" starts in the first column.
// All whitespace left of that column is removed from the string.

string rawStringLiteralDelimiter = """"
    Raw string literals are delimited
    by a string of at least three double quotes,
    like this: """
    """";
```

The following examples demonstrate the compiler errors reported based on these rules:

C#

```
// CS8997: Unterminated raw string literal.
var multiLineStart = """This
    is the beginning of a string
    """;

// CS9000: Raw string literal delimiter must be on its own line.
var multiLineEnd = """
    This is the beginning of a string """;

// CS8999: Line does not start with the same whitespace as the closing line
// of the raw string literal
var noOutdenting = """
    A line of text.
Trying to outdent the second line.
    """;
```

The first two examples are invalid because multiline raw string literals require the opening and closing quote sequence on its own line. The third example is invalid because the text is outdented from the closing quote sequence.

You should consider raw string literals when you're generating text that includes characters that require escape sequences when using quoted string literals or verbatim string literals. Raw string literals are easier for you and others to read because it more closely resembles the output text. For example, consider the following code that includes a string of formatted JSON:

C#

```
string jsonString = """
{
```

```
  "Date": "2019-08-01T00:00:00-07:00",
  "TemperatureCelsius": 25,
  "Summary": "Hot",
  "DatesAvailable": [
    "2019-08-01T00:00:00-07:00",
    "2019-08-02T00:00:00-07:00"
  ],
  "TemperatureRanges": {
    "Cold": {
      "High": 20,
      "Low": -10
    },
    "Hot": {
      "High": 60,
      "Low": 20
    }
            },
  "SummaryWords": [
    "Cool",
    "Windy",
    "Humid"
  ]
}
""";
```

Tip

Visual Studio and the C# Dev Kit provide some validation and syntax highlighting when raw string literals contain JSON data or regular expressions.

The tools parse the text. If the tools have confidence that the text represents JSON or a regular expression, the editor provides syntax coloring.

You can improve that experience by adding a comment above the declaration indicating the format:

- `// lang=json` indicates the raw string literal represents JSON data.
- `// lang=regex` indicates the raw string literal represents a regular expression.

When a raw string literal is used as an argument where the parameter uses the System.Diagnostics.CodeAnalysis.StringSyntaxAttribute to indicate a format, these tools validate the raw string literal for some of the format types. Both JSON and regex are supported.

For some formats, the comment or the attribute enables code suggestions offer fixes for string literals based on the format.

String escape sequences

Expand table

Escape sequence	Character name	Unicode encoding
`\'`	Single quote	0x0027
`\"`	Double quote	0x0022
`\\`	Backslash	0x005C
`\0`	Null	0x0000
`\a`	Alert	0x0007
`\b`	Backspace	0x0008
`\e`	Escape	0x001B
`\f`	Form feed	0x000C
`\n`	New line	0x000A
`\r`	Carriage return	0x000D
`\t`	Horizontal tab	0x0009
`\v`	Vertical tab	0x000B
`\u`	Unicode escape sequence (UTF-16)	`\uHHHH` (range: 0000 - FFFF; example: `\u00E7` = "ç")
`\U`	Unicode escape sequence (UTF-32)	`\U00HHHHHH` (range: 000000 - 10FFFF; example: `\U0001F47D` = "👽")
`\x`	Unicode escape sequence similar to "\u" except with variable length	`\xH[H][H][H]` (range: 0 - FFFF; example: `\x00E7` or `\x0E7` or `\xE7` = "ç")

> **⚠ Warning**
>
> When you use the `\x` escape sequence and specifying less than 4 hex digits, if the characters that immediately follow the escape sequence are valid hex digits (such as 0-9, A-F, and a-f), they're interpreted as being part of the escape sequence. For example, `\xA1` produces "¡", which is code point U+00A1. However, if the next character is "A" or "a", then the escape sequence will instead be interpreted as being `\xA1A` and produce "ਚ",

which is code point U+0A1A. In such cases, specifying all 4 hex digits (for example, `\x00A1`) prevents any possible misinterpretation.

> **Note**
>
> At compile time, verbatim and raw strings are converted to ordinary strings with all the same escape sequences. Therefore, if you view a verbatim or raw string in the debugger watch window, you see the escape characters added by the compiler, not the verbatim, or raw version from your source code. For example, the verbatim string `@"C:\files.txt"` appears in the watch window as `"C:\\files.txt"`.

Format strings

A format string is a string whose contents are determined dynamically at run time. Format strings are created by embedding *interpolated expressions* or placeholders inside of braces within a string. Everything inside the braces (`{...}`) is resolved to a value and output as a formatted string at run time. There are two methods to create format strings: string interpolation and composite formatting.

String interpolation

You declare *Interpolated strings* with the `$` special character. An interpolated string includes interpolated expressions in braces. If you're new to string interpolation, see the String interpolation - C# tutorial for several examples.

Use string interpolation to improve the readability and maintainability of your code. String interpolation achieves the same results as the `String.Format` method, but is easier to use and improves inline clarity.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($"{jh.firstName} {jh.lastName} was an African American poet born
in {jh.born}.");
Console.WriteLine($"He was first published in {jh.published} at the age of
{jh.published - jh.born}.");
Console.WriteLine($"He'd be over {Math.Round((2018d - jh.born) / 100d) * 100d} years
old today.");

// Output:
// Jupiter Hammon was an African American poet born in 1711.
```

```
// He was first published in 1761 at the age of 50.
// He'd be over 300 years old today.
```

You can use string interpolation to initialize a constant string when all the expressions used for placeholders are also constant strings.

You can combine *raw string literals* with string interpolations. You start and end the format string with three or more successive double quotes. If your output string should contain the `{` or `}` character, you can use extra `$` characters to specify how many `{` and `}` characters start and end an interpolation. Any sequence of fewer `{` or `}` characters is included in the output. The following example shows how you can use that feature to display the distance of a point from the origin, and place the point inside braces:

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y * Y)}}
from the origin.""";

Console.WriteLine(pointMessage);
// Output:
// The point {2, 3} is 3.605551275463989 from the origin.
```

Verbatim string interpolation

C# also allows verbatim string interpolation, for example across multiple lines, using the `$@` or `@$` syntax.

To interpret escape sequences literally, use a verbatim string literal. An interpolated verbatim string starts with the `$` character followed by the `@` character. You can use the `$` and `@` tokens in any order: both `$@"..."` and `@$"..."` are valid interpolated verbatim strings.

C#

```
var jh = (firstName: "Jupiter", lastName: "Hammon", born: 1711, published: 1761);
Console.WriteLine($@"{jh.firstName} {jh.lastName}
    was an African American poet born in {jh.born}.");
Console.WriteLine(@$"He was first published in {jh.published}
at the age of {jh.published - jh.born}.");

// Output:
// Jupiter Hammon
//     was an African American poet born in 1711.
```

```
// He was first published in 1761
// at the age of 50.
```

Composite formatting

The String.Format utilizes placeholders in braces to create a format string. This example results in similar output to the string interpolation method used in the preceding sample.

C#

```
var pw = (firstName: "Phillis", lastName: "Wheatley", born: 1753, published: 1773);
Console.WriteLine("{0} {1} was an African American poet born in {2}.", pw.firstName,
pw.lastName, pw.born);
Console.WriteLine("She was first published in {0} at the age of {1}.", pw.published,
pw.published - pw.born);
Console.WriteLine("She'd be over {0} years old today.", Math.Round((2018d - pw.born)
/ 100d) * 100d);

// Output:
// Phillis Wheatley was an African American poet born in 1753.
// She was first published in 1773 at the age of 20.
// She'd be over 300 years old today.
```

For more information, see Composite formatting in .NET.

Substrings

A substring is any sequence of characters that is contained in a string. Use the Substring method to create a new string from a part of the original string. You can search for one or more occurrences of a substring by using the IndexOf method. Use the Replace method to replace all occurrences of a specified substring with a new string. Like the Substring method, Replace actually returns a new string and doesn't modify the original string. For more information, see How to search strings and How to modify string contents.

C#

```
string s3 = "Visual C# Express";
System.Console.WriteLine(s3.Substring(7, 2));
// Output: "C#"

System.Console.WriteLine(s3.Replace("C#", "Basic"));
// Output: "Visual Basic Express"

// Index values are zero-based
int index = s3.IndexOf("C");
// index = 7
```

Accessing individual characters

You can use array notation with an index value to acquire read-only access to individual characters, as in the following example:

C#

```
string s5 = "Printing backwards";

for (int i = 0; i < s5.Length; i++)
{
    System.Console.Write(s5[s5.Length - i - 1]);
}
// Output: "sdrawkcab gnitnirP"
```

For more extensive string manipulation, you can use a StringBuilder object to modify the individual chars "in-place." Then create a new string to store the results by using the StringBuilder methods. In the following example, assume that you must modify the original string in a particular way and then store the results for future use:

C#

```
string question = "hOW DOES mICROSOFT wORD DEAL WITH THE cAPS lOCK KEY?";
System.Text.StringBuilder sb = new System.Text.StringBuilder(question);

for (int j = 0; j < sb.Length; j++)
{
    if (System.Char.IsLower(sb[j]) == true)
        sb[j] = System.Char.ToUpper(sb[j]);
    else if (System.Char.IsUpper(sb[j]) == true)
        sb[j] = System.Char.ToLower(sb[j]);
}
// Store the new string.
string corrected = sb.ToString();
System.Console.WriteLine(corrected);
// Output: How does Microsoft Word deal with the Caps Lock key?
```

For more information about indices, see the Explore indexes and ranges article.

Null strings and empty strings

An empty string is an instance of a System.String object that contains zero characters. Empty strings are used often in various programming scenarios to represent a blank text field. You can call methods on empty strings because they're valid System.String objects. Empty strings are initialized as follows:

C#

```
string s = String.Empty;
```

By contrast, a null string doesn't refer to an instance of a System.String object and any attempt to call a method on a null string causes a NullReferenceException. However, you can use null strings in concatenation and comparison operations with other strings. The following examples illustrate some cases in which a reference to a null string does and doesn't cause an exception to be thrown:

C#

```
string str = "hello";
string? nullStr = null;
string emptyStr = String.Empty;

string tempStr = str + nullStr;
// Output of the following line: hello
Console.WriteLine(tempStr);

bool b = (emptyStr == nullStr);
// Output of the following line: False
Console.WriteLine(b);

// The following line creates a new empty string.
string newStr = emptyStr + nullStr;

// Null strings and empty strings behave differently. The following
// two lines display 0.
Console.WriteLine(emptyStr.Length);
Console.WriteLine(newStr.Length);
// The following line raises a NullReferenceException.
//Console.WriteLine(nullStr.Length);

// The null character can be displayed and counted, like other chars.
string s1 = "\x0" + "abc";
string s2 = "abc" + "\x0";
// Output of the following line: * abc*
Console.WriteLine("*" + s1 + "*");
// Output of the following line: *abc *
Console.WriteLine("*" + s2 + "*");
// Output of the following line: 4
Console.WriteLine(s2.Length);
```

Using StringBuilder for fast string creation

String operations in .NET are highly optimized and in most cases don't significantly impact performance. However, in some scenarios such as tight loops that are executing many hundreds or thousands of times, string operations can affect performance. The StringBuilder

class creates a string buffer that offers better performance if your program performs many string manipulations. The StringBuilder string also enables you to reassign individual characters, something the built-in string data type doesn't support. This code, for example, changes the content of a string without creating a new string:

C#

```
System.Text.StringBuilder sb = new System.Text.StringBuilder("Rat: the ideal pet");
sb[0] = 'C';
System.Console.WriteLine(sb.ToString());
//Outputs Cat: the ideal pet
```

In this example, a StringBuilder object is used to create a string from a set of numeric types:

C#

```
var sb = new StringBuilder();

// Create a string composed of numbers 0 - 9
for (int i = 0; i < 10; i++)
{
    sb.Append(i.ToString());
}
Console.WriteLine(sb);  // displays 0123456789

// Copy one character of the string (not possible with a System.String)
sb[0] = sb[9];

Console.WriteLine(sb);  // displays 9123456789
```

Strings, extension methods, and LINQ

Because the String type implements IEnumerable<T>, you can use the extension methods defined in the Enumerable class on strings. To avoid visual clutter, these methods are excluded from IntelliSense for the String type, but they're available nevertheless. You can also use LINQ query expressions on strings. For more information, see LINQ and Strings.

Related articles

- How to modify string contents: Illustrates techniques to transform strings and modify the contents of strings.
- How to compare strings: Shows how to perform ordinal and culture specific comparisons of strings.
- How to concatenate multiple strings: Demonstrates various ways to join multiple strings into one.

- How to parse strings using String.Split: Contains code examples that illustrate how to use the String.Split method to parse strings.
- How to search strings: Explains how to use search for specific text or patterns in strings.
- How to determine whether a string represents a numeric value: Shows how to safely parse a string to see whether it has a valid numeric value.
- String interpolation: Describes the string interpolation feature that provides a convenient syntax to format strings.
- Using the StringBuilder class in .NET
- Parsing Strings: Describes how to convert string representations of .NET base types to instances of the corresponding types.
- Parsing Date and Time Strings in .NET: Shows how to convert a string such as "01/24/2008" to a System.DateTime object.
- Comparing Strings: Includes information about how to compare strings and provides examples in C# and Visual Basic.
- Using the StringBuilder Class: Describes how to create and modify dynamic string objects by using the StringBuilder class.
- LINQ and Strings: Provides information about how to perform various string operations by using LINQ queries.

Last updated on 11/18/2025

How to determine whether a string represents a numeric value (C# Programming Guide)

Article • 04/16/2022

To determine whether a string is a valid representation of a specified numeric type, use the static `TryParse` method that is implemented by all primitive numeric types and also by types such as DateTime and IPAddress. The following example shows how to determine whether "108" is a valid int.

C#

```
int i = 0;
string s = "108";
bool result = int.TryParse(s, out i); //i now = 108
```

If the string contains nonnumeric characters or the numeric value is too large or too small for the particular type you have specified, `TryParse` returns false and sets the out parameter to zero. Otherwise, it returns true and sets the out parameter to the numeric value of the string.

> **Note**
>
> A string may contain only numeric characters and still not be valid for the type whose `TryParse` method that you use. For example, "256" is not a valid value for `byte` but it is valid for `int`. "98.6" is not a valid value for `int` but it is a valid `decimal`.

Example

The following examples show how to use `TryParse` with string representations of `long`, `byte`, and `decimal` values.

C#

```
string numString = "1287543"; //"1287543.0" will return false for a long
long number1 = 0;
bool canConvert = long.TryParse(numString, out number1);
```

```
if (canConvert == true)
Console.WriteLine($"number1 now = {number1}");
else
Console.WriteLine("numString is not a valid long");

byte number2 = 0;
numString = "255"; // A value of 256 will return false
canConvert = byte.TryParse(numString, out number2);
if (canConvert == true)
Console.WriteLine($"number2 now = {number2}");
else
Console.WriteLine("numString is not a valid byte");

decimal number3 = 0;
numString = "27.3"; //"27" is also a valid decimal
canConvert = decimal.TryParse(numString, out number3);
if (canConvert == true)
Console.WriteLine($"number3 now = {number3}");
else
Console.WriteLine("number3 is not a valid decimal");
```

Robust Programming

Primitive numeric types also implement the `Parse` static method, which throws an exception if the string is not a valid number. `TryParse` is generally more efficient because it just returns false if the number is not valid.

.NET Security

Always use the `TryParse` or `Parse` methods to validate user input from controls such as text boxes and combo boxes.

See also

- How to convert a byte array to an int
- How to convert a string to a number
- How to convert between hexadecimal strings and numeric types
- Parsing Numeric Strings
- Formatting Types

Indexers

08/15/2025

You define *indexers* when instances of a class or struct can be indexed like an array or other collection. The indexed value can be set or retrieved without explicitly specifying a type or instance member. Indexers resemble properties except that their accessors take parameters.

The following example defines a generic class with get and set accessor methods to assign and retrieve values.

C#

```
namespace Indexers;

public class SampleCollection<T>
{
   // Declare an array to store the data elements.
   private T[] arr = new T[100];

   // Define the indexer to allow client code to use [] notation.
   public T this[int i]
   {
      get => arr[i];
      set => arr[i] = value;
   }
}
```

The preceding example shows a read / write indexer. It contains both the `get` and `set` accessors. You can define read only indexers as an expression bodied member, as shown in the following examples:

C#

```
namespace Indexers;

public class ReadOnlySampleCollection<T>(params IEnumerable<T> items)
{
   // Declare an array to store the data elements.
   private T[] arr = [.. items];

   public T this[int i] => arr[i];

}
```

The `get` keyword isn't used; `=>` introduces the expression body.

Indexers enable *indexed* properties: properties referenced using one or more arguments. Those arguments provide an index into some collection of values.

- Indexers enable objects to be indexed similar to arrays.
- A `get` accessor returns a value. A `set` accessor assigns a value.
- The this keyword defines the indexer.
- The value keyword is the argument to the `set` accessor.
- Indexers don't require an integer index value; it's up to you how to define the specific look-up mechanism.
- Indexers can be overloaded.
- Indexers can have one or more formal parameters, for example, when accessing a two-dimensional array.
- You can declare partial indexers in partial types.

You can apply almost everything you learned from working with properties to indexers. The only exception to that rule is *automatically implemented properties*. The compiler can't always generate the correct storage for an indexer. You can define multiple indexers on a type, as long as the argument lists for each indexer is unique.

Uses of indexers

You define *indexers* in your type when its API models some collection. Your indexer isn't required to map directly to the collection types that are part of the .NET core framework. Indexers enable you to provide the API that matches your type's abstraction without exposing the inner details of how the values for that abstraction are stored or computed.

Arrays and Vectors

Your type might model an array or a vector. The advantage of creating your own indexer is that you can define the storage for that collection to suit your needs. Imagine a scenario where your type models historical data that is too large to load into memory at once. You need to load and unload sections of the collection based on usage. The example following models this behavior. It reports on how many data points exist. It creates pages to hold sections of the data on demand. It removes pages from memory to make room for pages needed by more recent requests.

C#

```
namespace Indexers;

public record Measurements(double HiTemp, double LoTemp, double AirPressure);
```

```
public class DataSamples
{
    private class Page
    {
        private readonly List<Measurements> pageData = new ();
        private readonly int _startingIndex;
        private readonly int _length;

        public Page(int startingIndex, int length)
        {
            _startingIndex = startingIndex;
            _length = length;

            // This stays as random stuff:
            var generator = new Random();
            for (int i = 0; i < length; i++)
            {
                var m = new Measurements(HiTemp: generator.Next(50, 95),
                    LoTemp: generator.Next(12, 49),
                    AirPressure: 28.0 + generator.NextDouble() * 4
                );
                pageData.Add(m);
            }
        }
        public bool HasItem(int index) =>
            ((index >= _startingIndex) &&
            (index < _startingIndex + _length));

        public Measurements this[int index]
        {
            get
            {
                LastAccess = DateTime.Now;
                return pageData[index - _startingIndex];
            }
            set
            {
                pageData[index - _startingIndex] = value;
                Dirty = true;
                LastAccess = DateTime.Now;
            }
        }

        public bool Dirty { get; private set; } = false;
        public DateTime LastAccess { get; set; } = DateTime.Now;
    }

    private readonly int _totalSize;
    private readonly List<Page> pagesInMemory = new ();

    public DataSamples(int totalSize)
    {
        this._totalSize = totalSize;
    }
```

```
    public Measurements this[int index]
    {
        get
        {
            if (index < 0) throw new IndexOutOfRangeException("Cannot index less
than 0");
            if (index >= _totalSize) throw new IndexOutOfRangeException("Cannot
index past the end of storage");

            var page = updateCachedPagesForAccess(index);
            return page[index];
        }
        set
        {
            if (index < 0) throw new IndexOutOfRangeException("Cannot index less
than 0");
            if (index >= _totalSize) throw new IndexOutOfRangeException("Cannot
index past the end of storage");
            var page = updateCachedPagesForAccess(index);

            page[index] = value;
        }
    }

    private Page updateCachedPagesForAccess(int index)
    {
        foreach (var p in pagesInMemory)
        {
            if (p.HasItem(index))
            {
                return p;
            }
        }
        var startingIndex = (index / 1000) * 1000;
        var newPage = new Page(startingIndex, 1000);
        addPageToCache(newPage);
        return newPage;
    }

    private void addPageToCache(Page p)
    {
        if (pagesInMemory.Count > 4)
        {
            // remove oldest non-dirty page:
            var oldest = pagesInMemory
                .Where(page => !page.Dirty)
                .OrderBy(page => page.LastAccess)
                .FirstOrDefault();
            // Note that this may keep more than 5 pages in memory
            // if too much is dirty
            if (oldest != null)
                pagesInMemory.Remove(oldest);
        }
        pagesInMemory.Add(p);
```

```
    }
}
```

You can follow this design idiom to model any sort of collection where there are good reasons not to load the entire set of data into an in-memory collection. Notice that the `Page` class is a private nested class that isn't part of the public interface. Those details are hidden from users of this class.

Dictionaries

Another common scenario is when you need to model a dictionary or a map. This scenario is when your type stores values based on key, possibly text keys. This example creates a dictionary that maps command line arguments to lambda expressions that manage those options. The following example shows two classes: an `ArgsActions` class that maps a command line option to an System.Action delegate, and an `ArgsProcessor` that uses the `ArgsActions` to execute each Action when it encounters that option.

C#

```
namespace Indexers;
public class ArgsProcessor
{
    private readonly ArgsActions _actions;

    public ArgsProcessor(ArgsActions actions)
    {
        _actions = actions;
    }

    public void Process(string[] args)
    {
        foreach (var arg in args)
        {
            _actions[arg]?.Invoke();
        }
    }

}
public class ArgsActions
{
    readonly private Dictionary<string, Action> _argsActions = new();

    public Action this[string s]
    {
        get
        {
            Action? action;
            Action defaultAction = () => { };
            return _argsActions.TryGetValue(s, out action) ? action :
```

```
defaultAction;
        }
    }

    public void SetOption(string s, Action a)
    {
        _argsActions[s] = a;
    }
}
```

In this example, the `ArgsAction` collection maps closely to the underlying collection. The `get` determines if a given option is configured. If so, it returns the Action associated with that option. If not, it returns an Action that does nothing. The public accessor doesn't include a `set` accessor. Rather, the design is using a public method for setting options.

Date-based indexers

When working with date-based data, you can use either DateTime or DateOnly as indexer keys. Use DateOnly when you only need the date part and want to avoid time-related complications. The following example shows a temperature tracking system that uses `DateOnly` as the primary indexer key:

C#

```
using System;
using System.Collections.Generic;

namespace Indexers;

public class DailyTemperatureData
{
    private readonly Dictionary<DateOnly, (double High, double Low)>
_temperatureData = new();

    // Indexer using DateOnly for date-only scenarios
    public (double High, double Low) this[DateOnly date]
    {
        get
        {
            if (_temperatureData.TryGetValue(date, out var temp))
            {
                return temp;
            }
            throw new KeyNotFoundException($"No temperature data available for
{date:yyyy-MM-dd}");
        }
        set
        {
            _temperatureData[date] = value;
```

```
        }
    }

    // Overload using DateTime for convenience, but only uses the date part
    public (double High, double Low) this[DateTime dateTime]
    {
        get => this[DateOnly.FromDateTime(dateTime)];
        set => this[DateOnly.FromDateTime(dateTime)] = value;
    }

    public bool HasDataFor(DateOnly date) => _temperatureData.ContainsKey(date);

    public IEnumerable<DateOnly> AvailableDates => _temperatureData.Keys;
}
```

This example demonstrates both `DateOnly` and `DateTime` indexers. While the `DateOnly` indexer is the primary interface, the `DateTime` overload provides convenience by extracting only the date portion. This approach ensures that all temperature data for a given day is treated consistently, regardless of the time component.

Multi-Dimensional Maps

You can create indexers that use multiple arguments. In addition, those arguments aren't constrained to be the same type.

The following example shows a class that generates values for a Mandelbrot set. For more information on the mathematics behind the set, read this article. The indexer uses two doubles to define a point in the X, Y plane. The `get` accessor computes the number of iterations until a point is determined to be not in the set. When the maximum number of iterations is reached, the point is in the set, and the class's maxIterations value is returned. (The computer generated images popularized for the Mandelbrot set define colors for the number of iterations necessary to determine that a point is outside the set.)

C#

```
namespace Indexers;
public class Mandelbrot(int maxIterations)
{

    public int this[double x, double y]
    {
        get
        {
            var iterations = 0;
            var x0 = x;
            var y0 = y;

            while ((x * x + y * y < 4) &&
```

```
                (iterations < maxIterations))
            {
                (x, y) = (x * x - y * y + x0, 2 * x * y + y0);
                iterations++;
            }
            return iterations;
        }
    }
}
```

The Mandelbrot Set defines values at every (x,y) coordinate for real number values. That defines a dictionary that could contain an infinite number of values. Therefore, there's no storage behind the set. Instead, this class computes the value for each point when code calls the `get` accessor. There's no underlying storage used.

Summing Up

You create indexers anytime you have a property-like element in your class where that property represents not a single value, but rather a set of values. One or more arguments identify each individual item. Those arguments can uniquely identify which item in the set should be referenced. Indexers extend the concept of properties, where a member is treated like a data item from outside the class, but like a method on the inside. Indexers allow arguments to find a single item in a property that represents a set of items.

You can access the sample folder for indexers. For download instructions, see Samples and Tutorials.

C# Language Specification

For more information, see Indexers in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

Using indexers (C# Programming Guide)

Article • 08/23/2024

Indexers are a syntactic convenience that enables you to create a class, struct, or interface that client applications can access as an array. The compiler generates an `Item` property (or an alternatively named property if IndexerNameAttribute is present), and the appropriate accessor methods. Indexers are most frequently implemented in types whose primary purpose is to encapsulate an internal collection or array. For example, suppose you have a class `TempRecord` that represents the temperature in Fahrenheit as recorded at 10 different times during a 24-hour period. The class contains a `temps` array of type `float[]` to store the temperature values. By implementing an indexer in this class, clients can access the temperatures in a `TempRecord` instance as `float temp = tempRecord[4]` instead of as `float temp = tempRecord.temps[4]`. The indexer notation not only simplifies the syntax for client applications; it also makes the class, and its purpose more intuitive for other developers to understand.

To declare an indexer on a class or struct, use the this keyword, as the following example shows:

C#

```
// Indexer declaration
public int this[int index]
{
    // get and set accessors
}
```

> **ⓘ Important**
>
> Declaring an indexer will automatically generate a property named `Item` on the object. The `Item` property is not directly accessible from the instance member access expression. Additionally, if you add your own `Item` property to an object with an indexer, you'll get a CS0102 compiler error. To avoid this error, use the IndexerNameAttribute rename the indexer as detailed later in this article.

Remarks

The type of an indexer and the type of its parameters must be at least as accessible as the indexer itself. For more information about accessibility levels, see Access Modifiers.

For more information about how to use indexers with an interface, see Interface Indexers.

The signature of an indexer consists of the number and types of its formal parameters. It doesn't include the indexer type or the names of the formal parameters. If you declare more than one indexer in the same class, they must have different signatures.

An indexer isn't classified as a variable; therefore, an indexer value can't be passed by reference (as a ref or out parameter) unless its value is a reference (that is, it returns by reference.)

To provide the indexer with a name that other languages can use, use System.Runtime.CompilerServices.IndexerNameAttribute, as the following example shows:

```
C#

// Indexer declaration
[System.Runtime.CompilerServices.IndexerName("TheItem")]
public int this[int index]
{
    // get and set accessors
}
```

This indexer has the name `TheItem`, as it's overridden by the indexer name attribute. By default, the indexer name is `Item`.

Example 1

The following example shows how to declare a private array field, `temps`, and an indexer. The indexer enables direct access to the instance `tempRecord[i]`. The alternative to using the indexer is to declare the array as a public member and access its members, `tempRecord.temps[i]`, directly.

```
C#

public class TempRecord
{
    // Array of temperature values
    float[] temps =
    [
        56.2F, 56.7F, 56.5F, 56.9F, 58.8F,
        61.3F, 65.9F, 62.1F, 59.2F, 57.5F
    ];

    // To enable client code to validate input
```

```
    // when accessing your indexer.
    public int Length => temps.Length;

    // Indexer declaration.
    // If index is out of range, the temps array will throw the exception.
    public float this[int index]
    {
        get => temps[index];
        set => temps[index] = value;
    }
}
```

Notice that when an indexer's access is evaluated, for example, in a `Console.Write` statement, the get accessor is invoked. Therefore, if no `get` accessor exists, a compile-time error occurs.

C#

```
var tempRecord = new TempRecord();

// Use the indexer's set accessor
tempRecord[3] = 58.3F;
tempRecord[5] = 60.1F;

// Use the indexer's get accessor
for (int i = 0; i < 10; i++)
{
    Console.WriteLine($"Element #{i} = {tempRecord[i]}");
}
```

Indexing using other values

C# doesn't limit the indexer parameter type to integer. For example, it can be useful to use a string with an indexer. Such an indexer might be implemented by searching for the string in the collection, and returning the appropriate value. As accessors can be overloaded, the string and integer versions can coexist.

Example 2

The following example declares a class that stores the days of the week. A `get` accessor takes a string, the name of a day, and returns the corresponding integer. For example, "Sunday" returns 0, "Monday" returns 1, and so on.

C#

```
// Using a string as an indexer value
class DayCollection
{
    string[] days = ["Sun", "Mon", "Tues", "Wed", "Thurs", "Fri", "Sat"];

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[string day] => FindDayIndex(day);

    private int FindDayIndex(string day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }

        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be in the form
\"Sun\", \"Mon\", etc");
    }
}
```

Consuming example 2

C#

```
var week = new DayCollection();
Console.WriteLine(week["Fri"]);

try
{
    Console.WriteLine(week["Made-up day"]);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

Example 3

The following example declares a class that stores the days of the week using the System.DayOfWeek enum. A `get` accessor takes a `DayOfWeek`, the value of a day, and

returns the corresponding integer. For example, `DayOfWeek.Sunday` returns 0, `DayOfWeek.Monday` returns 1, and so on.

C#

```
using Day = System.DayOfWeek;

class DayOfWeekCollection
{
    Day[] days =
    [
        Day.Sunday, Day.Monday, Day.Tuesday, Day.Wednesday,
        Day.Thursday, Day.Friday, Day.Saturday
    ];

    // Indexer with only a get accessor with the expression-bodied
definition:
    public int this[Day day] => FindDayIndex(day);

    private int FindDayIndex(Day day)
    {
        for (int j = 0; j < days.Length; j++)
        {
            if (days[j] == day)
            {
                return j;
            }
        }
        throw new ArgumentOutOfRangeException(
            nameof(day),
            $"Day {day} is not supported.\nDay input must be a defined
System.DayOfWeek value.");
    }
}
```

Consuming example 3

C#

```
var week = new DayOfWeekCollection();
Console.WriteLine(week[DayOfWeek.Friday]);

try
{
    Console.WriteLine(week[(DayOfWeek)43]);
}
catch (ArgumentOutOfRangeException e)
{
    Console.WriteLine($"Not supported input: {e.Message}");
}
```

Robust programming

There are two main ways in which the security and reliability of indexers can be improved:

- Be sure to incorporate some type of error-handling strategy to handle the chance of client code passing in an invalid index value. In the first example earlier in this article, the TempRecord class provides a Length property that enables the client code to verify the input before passing it to the indexer. You can also put the error handling code inside the indexer itself. Be sure to document for users any exceptions that you throw inside an indexer accessor.
- Set the accessibility of the get and set accessors to be as restrictive as is reasonable. This is important for the `set` accessor in particular. For more information, see Restricting Accessor Accessibility.

See also

- Indexers
- Properties

Indexers in Interfaces (C# Programming Guide)

Article • 08/23/2024

Indexers can be declared on an interface. Accessors of interface indexers differ from the accessors of class indexers in the following ways:

- Interface accessors don't use modifiers.
- An interface accessor typically doesn't have a body.

The purpose of the accessor is to indicate whether the indexer is read-write, read-only, or write-only. You can provide an implementation for an indexer defined in an interface, but this is rare. Indexers typically define an API to access data fields, and data fields can't be defined in an interface.

The following is an example of an interface indexer accessor:

C#

```
public interface ISomeInterface
{
    //...

    // Indexer declaration:
    string this[int index]
    {
        get;
        set;
    }
}
```

The signature of an indexer must differ from the signatures of all other indexers declared in the same interface.

Example

The following example shows how to implement interface indexers.

C#

```
// Indexer on an interface:
public interface IIndexInterface
{
    // Indexer declaration:
```

```
    int this[int index]
    {
        get;
        set;
    }
}

// Implementing the interface.
class IndexerClass : IIndexInterface
{
    private int[] arr = new int[100];
    public int this[int index]   // indexer declaration
    {
        // The arr object will throw IndexOutOfRange exception.
        get => arr[index];
        set => arr[index] = value;
    }
}
```

C#

```
IndexerClass test = new IndexerClass();
System.Random rand = System.Random.Shared;
// Call the indexer to initialize its elements.
for (int i = 0; i < 10; i++)
{
    test[i] = rand.Next();
}
for (int i = 0; i < 10; i++)
{
    System.Console.WriteLine($"Element #{i} = {test[i]}");
}

/* Sample output:
    Element #0 = 360877544
    Element #1 = 327058047
    Element #2 = 1913480832
    Element #3 = 1519039937
    Element #4 = 601472233
    Element #5 = 323352310
    Element #6 = 1422639981
    Element #7 = 1797892494
    Element #8 = 875761049
    Element #9 = 393083859
*/
```

In the preceding example, you could use the explicit interface member implementation by using the fully qualified name of the interface member. For example

C#

```
string IIndexInterface.this[int index]
{
}
```

However, the fully qualified name is only needed to avoid ambiguity when the class is implementing more than one interface with the same indexer signature. For example, if an `Employee` class is implementing two interfaces, `ICitizen` and `IEmployee`, and both interfaces have the same indexer signature, the explicit interface member implementation is necessary. That is, the following indexer declaration:

C#

```
string IEmployee.this[int index]
{
}
```

Implements the indexer on the `IEmployee` interface, while the following declaration:

C#

```
string ICitizen.this[int index]
{
}
```

Implements the indexer on the `ICitizen` interface.

See also

- Indexers
- Properties
- Interfaces

Comparison Between Properties and Indexers (C# Programming Guide)

Article • 03/12/2024

Indexers are like properties. Except for the differences shown in the following table, all the rules that are defined for property accessors apply to indexer accessors also.

Expand table

Property	Indexer
Allows methods to be called as if they were public data members.	Allows elements of an internal collection of an object to be accessed by using array notation on the object itself.
Accessed through a simple name.	Accessed through an index.
Can be a static or an instance member.	Must be an instance member.
A get accessor of a property has no parameters.	A `get` accessor of an indexer has the same formal parameter list as the indexer.
A set accessor of a property contains the implicit `value` parameter.	A `set` accessor of an indexer has the same formal parameter list as the indexer, and also to the value parameter.
Supports shortened syntax with Automatically implemented properties.	Supports expression bodied members for get only indexers.

See also

- Indexers
- Properties

Events (C# Programming Guide)

Article • 03/11/2025

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*.

In a typical C# Windows Forms or Web application, you subscribe to events raised by controls such as buttons and list boxes. You can use the Visual C# integrated development environment (IDE) to browse the events that a control publishes and select the ones that you want to handle. The IDE provides an easy way to automatically add an empty event handler method and the code to subscribe to the event. For more information, see How to subscribe to and unsubscribe from events.

Events Overview

Events have the following properties:

- The publisher determines when an event is raised; the subscribers determine what action is taken in response to the event.
- An event can have multiple subscribers. A subscriber can handle multiple events from multiple publishers.
- Events that have no subscribers are never raised.
- Events are typically used to signal user actions such as button clicks or menu selections in graphical user interfaces.
- When an event has multiple subscribers, the event handlers are invoked synchronously when an event is raised. To invoke events asynchronously, see Calling Synchronous Methods Asynchronously.
- In the .NET class library, events are based on the EventHandler delegate and the EventArgs base class.

Related Sections

For more information, see:

- How to subscribe to and unsubscribe from events
- How to publish events that conform to .NET Guidelines
- How to raise base class events in derived classes
- How to implement interface events
- How to implement custom event accessors

C# Language Specification

For more information, see Events in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- EventHandler
- Delegates
- Creating Event Handlers in Windows Forms

How to subscribe to and unsubscribe from events (C# Programming Guide)

Article • 10/12/2021

You subscribe to an event that is published by another class when you want to write custom code that is called when that event is raised. For example, you might subscribe to a button's `click` event in order to make your application do something useful when the user clicks the button.

To subscribe to events by using the Visual Studio IDE

1. If you cannot see the **Properties** window, in **Design** view, right-click the form or control for which you want to create an event handler, and select **Properties**.

2. On top of the **Properties** window, click the **Events** icon.

3. Double-click the event that you want to create, for example the `Load` event.

   Visual C# creates an empty event handler method and adds it to your code. Alternatively you can add the code manually in **Code** view. For example, the following lines of code declare an event handler method that will be called when the `Form` class raises the `Load` event.

   ```
   C#
   ```

   ```
   private void Form1_Load(object sender, System.EventArgs e)
   {
       // Add your form load event handling code here.
   }
   ```

   The line of code that is required to subscribe to the event is also automatically generated in the `InitializeComponent` method in the Form1.Designer.cs file in your project. It resembles this:

   ```
   C#
   ```

   ```
   this.Load += new System.EventHandler(this.Form1_Load);
   ```

To subscribe to events programmatically

1. Define an event handler method whose signature matches the delegate signature for the event. For example, if the event is based on the EventHandler delegate type, the following code represents the method stub:

   ```
   C#

   void HandleCustomEvent(object sender, CustomEventArgs a)
   {
      // Do something useful here.
   }
   ```

2. Use the addition assignment operator (`+=`) to attach an event handler to the event. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent`. Note that the subscriber class needs a reference to the publisher class in order to subscribe to its events.

   ```
   C#

   publisher.RaiseCustomEvent += HandleCustomEvent;
   ```

   You can also use a lambda expression to specify an event handler:

   ```
   C#

   public Form1()
   {
       InitializeComponent();
       this.Click += (s,e) =>
           {
               MessageBox.Show(((MouseEventArgs)e).Location.ToString());
           };
   }
   ```

To subscribe to events by using an anonymous function

If you don't have to unsubscribe from an event later, you can use the addition assignment operator (`+=`) to attach an anonymous function as an event handler. In the following example, assume that an object named `publisher` has an event named `RaiseCustomEvent` and that a `CustomEventArgs` class has also been defined to carry some kind of specialized event information. Note that the subscriber class needs a reference to `publisher` in order to subscribe to its events.

```
C#
```

```
publisher.RaiseCustomEvent += (object o, CustomEventArgs e) =>
{
  string s = o.ToString() + " " + e.ToString();
  Console.WriteLine(s);
};
```

You cannot easily unsubscribe from an event if you used an anonymous function to subscribe to it. To unsubscribe in this scenario, go back to the code where you subscribe to the event, store the anonymous function in a delegate variable, and then add the delegate to the event. We recommend that you don't use anonymous functions to subscribe to events if you have to unsubscribe from the event at some later point in your code. For more information about anonymous functions, see Lambda expressions.

Unsubscribing

To prevent your event handler from being invoked when the event is raised, unsubscribe from the event. In order to prevent resource leaks, you should unsubscribe from events before you dispose of a subscriber object. Until you unsubscribe from an event, the multicast delegate that underlies the event in the publishing object has a reference to the delegate that encapsulates the subscriber's event handler. As long as the publishing object holds that reference, garbage collection will not delete your subscriber object.

To unsubscribe from an event

- Use the subtraction assignment operator (`-=`) to unsubscribe from an event:

  ```
  C#

  publisher.RaiseCustomEvent -= HandleCustomEvent;
  ```

  When all subscribers have unsubscribed from an event, the event instance in the publisher class is set to `null`.

See also

- Events
- event
- How to publish events that conform to .NET Guidelines
- - and -= operators
- + and += operators

How to raise base class events in derived classes (C# Programming Guide)

Article • 03/12/2024

The following simple example shows the standard way to declare events in a base class so that they can also be raised from derived classes. This pattern is used extensively in Windows Forms classes in the .NET class libraries.

When you create a class that can be used as a base class for other classes, you should consider the fact that events are a special type of delegate that can only be invoked from within the class that declared them. Derived classes cannot directly invoke events that are declared within the base class. Although sometimes you may want an event that can only be raised by the base class, most of the time, you should enable the derived class to invoke base class events. To do this, you can create a protected invoking method in the base class that wraps the event. By calling or overriding this invoking method, derived classes can invoke the event indirectly.

> **Note**
>
> Do not declare virtual events in a base class and override them in a derived class. The C# compiler does not handle these correctly and it is unpredictable whether a subscriber to the derived event will actually be subscribing to the base class event.

Example

C#

```
namespace BaseClassEvents
{
    // Special EventArgs class to hold info about Shapes.
    public class ShapeEventArgs : EventArgs
    {
        public ShapeEventArgs(double area)
        {
            NewArea = area;
        }

        public double NewArea { get; }
    }

    // Base class event publisher
```

```
    public abstract class Shape
    {
        protected double _area;

        public double Area
        {
            get => _area;
            set => _area = value;
        }

        // The event. Note that by using the generic EventHandler<T> event
type
        // we do not need to declare a separate delegate type.
        public event EventHandler<ShapeEventArgs> ShapeChanged;

        public abstract void Draw();

        //The event-invoking method that derived classes can override.
        protected virtual void OnShapeChanged(ShapeEventArgs e)
        {
            // Safely raise the event for all subscribers
            ShapeChanged?.Invoke(this, e);
        }
    }

    public class Circle : Shape
    {
        private double _radius;

        public Circle(double radius)
        {
            _radius = radius;
            _area = 3.14 * _radius * _radius;
        }

        public void Update(double d)
        {
            _radius = d;
            _area = 3.14 * _radius * _radius;
            OnShapeChanged(new ShapeEventArgs(_area));
        }

        protected override void OnShapeChanged(ShapeEventArgs e)
        {
            // Do any circle-specific processing here.

            // Call the base class event invocation method.
            base.OnShapeChanged(e);
        }

        public override void Draw()
        {
            Console.WriteLine("Drawing a circle");
        }
    }
```

```
public class Rectangle : Shape
{
    private double _length;
    private double _width;

    public Rectangle(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
    }

    public void Update(double length, double width)
    {
        _length = length;
        _width = width;
        _area = _length * _width;
        OnShapeChanged(new ShapeEventArgs(_area));
    }

    protected override void OnShapeChanged(ShapeEventArgs e)
    {
        // Do any rectangle-specific processing here.

        // Call the base class event invocation method.
        base.OnShapeChanged(e);
    }

    public override void Draw()
    {
        Console.WriteLine("Drawing a rectangle");
    }
}

// Represents the surface on which the shapes are drawn
// Subscribes to shape events so that it knows
// when to redraw a shape.
public class ShapeContainer
{
    private readonly List<Shape> _list;

    public ShapeContainer()
    {
        _list = new List<Shape>();
    }

    public void AddShape(Shape shape)
    {
        _list.Add(shape);

        // Subscribe to the base class event.
        shape.ShapeChanged += HandleShapeChanged;
    }
```

```
        // ...Other methods to draw, resize, etc.

        private void HandleShapeChanged(object sender, ShapeEventArgs e)
        {
            if (sender is Shape shape)
            {
                // Diagnostic message for demonstration purposes.
                Console.WriteLine($"Received event. Shape area is now
{e.NewArea}");

                // Redraw the shape here.
                shape.Draw();
            }
        }
    }

    class Test
    {
        static void Main()
        {
            //Create the event publishers and subscriber
            var circle = new Circle(54);
            var rectangle = new Rectangle(12, 9);
            var container = new ShapeContainer();

            // Add the shapes to the container.
            container.AddShape(circle);
            container.AddShape(rectangle);

            // Cause some events to be raised.
            circle.Update(57);
            rectangle.Update(7, 7);

            // Keep the console window open in debug mode.
            Console.WriteLine("Press any key to continue...");
            Console.ReadKey();
        }
    }
}
/* Output:
        Received event. Shape area is now 10201.86
        Drawing a circle
        Received event. Shape area is now 49
        Drawing a rectangle
 */
```

See also

- Events
- Delegates
- Access Modifiers

- Creating Event Handlers in Windows Forms

How to implement interface events (C# Programming Guide)

Article • 09/15/2021

An interface can declare an event. The following example shows how to implement interface events in a class. Basically the rules are the same as when you implement any interface method or property.

To implement interface events in a class

Declare the event in your class and then invoke it in the appropriate areas.

C#

```
namespace ImplementInterfaceEvents
{
    public interface IDrawingObject
    {
        event EventHandler ShapeChanged;
    }
    public class MyEventArgs : EventArgs
    {
        // class members
    }
    public class Shape : IDrawingObject
    {
        public event EventHandler ShapeChanged;
        void ChangeShape()
        {
            // Do something here before the event…

            OnShapeChanged(new MyEventArgs(/*arguments*/));

            // or do something here after the event.
        }
        protected virtual void OnShapeChanged(MyEventArgs e)
        {
            ShapeChanged?.Invoke(this, e);
        }
    }

}
```

Example

The following example shows how to handle the less-common situation in which your class inherits from two or more interfaces and each interface has an event with the same name. In this situation, you must provide an explicit interface implementation for at least one of the events. When you write an explicit interface implementation for an event, you must also write the `add` and `remove` event accessors. Normally these are provided by the compiler, but in this case the compiler cannot provide them.

By providing your own accessors, you can specify whether the two events are represented by the same event in your class, or by different events. For example, if the events should be raised at different times according to the interface specifications, you can associate each event with a separate implementation in your class. In the following example, subscribers determine which `OnDraw` event they will receive by casting the shape reference to either an `IShape` or an `IDrawingObject`.

C#

```
namespace WrapTwoInterfaceEvents
{
    using System;

    public interface IDrawingObject
    {
        // Raise this event before drawing
        // the object.
        event EventHandler OnDraw;
    }
    public interface IShape
    {
        // Raise this event after drawing
        // the shape.
        event EventHandler OnDraw;
    }

    // Base class event publisher inherits two
    // interfaces, each with an OnDraw event
    public class Shape : IDrawingObject, IShape
    {
        // Create an event for each interface event
        event EventHandler PreDrawEvent;
        event EventHandler PostDrawEvent;

        object objectLock = new Object();

        // Explicit interface implementation required.
        // Associate IDrawingObject's event with
        // PreDrawEvent
        #region IDrawingObjectOnDraw
        event EventHandler IDrawingObject.OnDraw
        {
            add
```

```
            {
                lock (objectLock)
                {
                    PreDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PreDrawEvent -= value;
                }
            }
        }
        #endregion
        // Explicit interface implementation required.
        // Associate IShape's event with
        // PostDrawEvent
        event EventHandler IShape.OnDraw
        {
            add
            {
                lock (objectLock)
                {
                    PostDrawEvent += value;
                }
            }
            remove
            {
                lock (objectLock)
                {
                    PostDrawEvent -= value;
                }
            }
        }

        // For the sake of simplicity this one method
        // implements both interfaces.
        public void Draw()
        {
            // Raise IDrawingObject's event before the object is drawn.
            PreDrawEvent?.Invoke(this, EventArgs.Empty);

            Console.WriteLine("Drawing a shape.");

            // Raise IShape's event after the object is drawn.
            PostDrawEvent?.Invoke(this, EventArgs.Empty);
        }
    }
    public class Subscriber1
    {
        // References the shape object as an IDrawingObject
        public Subscriber1(Shape shape)
        {
            IDrawingObject d = (IDrawingObject)shape;
```

```
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub1 receives the IDrawingObject event.");
        }
    }
    // References the shape object as an IShape
    public class Subscriber2
    {
        public Subscriber2(Shape shape)
        {
            IShape d = (IShape)shape;
            d.OnDraw += d_OnDraw;
        }

        void d_OnDraw(object sender, EventArgs e)
        {
            Console.WriteLine("Sub2 receives the IShape event.");
        }
    }

    public class Program
    {
        static void Main(string[] args)
        {
            Shape shape = new Shape();
            Subscriber1 sub = new Subscriber1(shape);
            Subscriber2 sub2 = new Subscriber2(shape);
            shape.Draw();

            // Keep the console window open in debug mode.
            System.Console.WriteLine("Press any key to exit.");
            System.Console.ReadKey();
        }
    }
}
/* Output:
    Sub1 receives the IDrawingObject event.
    Drawing a shape.
    Sub2 receives the IShape event.
*/
```

See also

- Events
- Delegates
- Explicit Interface Implementation
- How to raise base class events in derived classes

How to implement custom event accessors (C# Programming Guide)

Article • 09/15/2021

An event is a special kind of multicast delegate that can only be invoked from within the class that it is declared in. Client code subscribes to the event by providing a reference to a method that should be invoked when the event is fired. These methods are added to the delegate's invocation list through event accessors, which resemble property accessors, except that event accessors are named `add` and `remove`. In most cases, you do not have to supply custom event accessors. When no custom event accessors are supplied in your code, the compiler will add them automatically. However, in some cases you may have to provide custom behavior. One such case is shown in the topic How to implement interface events.

Example

The following example shows how to implement custom add and remove event accessors. Although you can substitute any code inside the accessors, we recommend that you lock the event before you add or remove a new event handler method.

C#

```
event EventHandler IDrawingObject.OnDraw
{
    add
    {
        lock (objectLock)
        {
            PreDrawEvent += value;
        }
    }
    remove
    {
        lock (objectLock)
        {
            PreDrawEvent -= value;
        }
    }
}
```

See also

- Events
- event

Generic type parameters (C# Programming Guide)

Article • 03/12/2024

In a generic type or method definition, a type parameter is a placeholder for a specific type that a client specifies when they create an instance of the generic type. A generic class, such as `GenericList<T>` listed in Introduction to Generics, cannot be used as-is because it is not really a type; it is more like a blueprint for a type. To use `GenericList<T>`, client code must declare and instantiate a constructed type by specifying a type argument inside the angle brackets. The type argument for this particular class can be any type recognized by the compiler. Any number of constructed type instances can be created, each one using a different type argument, as follows:

```
C#

GenericList<float> list1 = new GenericList<float>();
GenericList<ExampleClass> list2 = new GenericList<ExampleClass>();
GenericList<ExampleStruct> list3 = new GenericList<ExampleStruct>();
```

In each of these instances of `GenericList<T>`, every occurrence of `T` in the class is substituted at run time with the type argument. By means of this substitution, we have created three separate type-safe and efficient objects using a single class definition. For more information on how this substitution is performed by the CLR, see Generics in the Runtime.

You can learn the naming conventions for generic type parameters in the article on naming conventions.

See also

- System.Collections.Generic
- Generics
- Differences Between C++ Templates and C# Generics

Constraints on type parameters (C# Programming Guide)

Constraints inform the compiler about the capabilities a type argument must have. Without any constraints, the type argument could be any type. The compiler can only assume the members of System.Object, which is the ultimate base class for any .NET type. For more information, see Why use constraints. If client code uses a type that doesn't satisfy a constraint, the compiler issues an error. Constraints are specified by using the `where` contextual keyword. The following table lists the various types of constraints:

Expand table

Constraint	Description
`where T : struct`	The type argument must be a non-nullable value type, which includes `record struct` types. For information about nullable value types, see Nullable value types. Because all value types have an accessible parameterless constructor, either declared or implicit, the `struct` constraint implies the `new()` constraint and can't be combined with the `new()` constraint. You can't combine the `struct` constraint with the `unmanaged` constraint.
`where T : class`	The type argument must be a reference type. This constraint applies also to any class, interface, delegate, or array type. In a nullable context, `T` must be a non-nullable reference type.
`where T : class?`	The type argument must be a reference type, either nullable or non-nullable. This constraint applies also to any class, interface, delegate, or array type, including records.
`where T : notnull`	The type argument must be a non-nullable type. The argument can be a non-nullable reference type or a non-nullable value type.
`where T : unmanaged`	The type argument must be a non-nullable unmanaged type. The `unmanaged` constraint implies the `struct` constraint and can't be combined with either the `struct` or `new()` constraints.
`where T : new()`	The type argument must have a public parameterless constructor. When used together with other constraints, the `new()` constraint must be specified last. The `new()` constraint can't be combined with the `struct` and `unmanaged` constraints.
`where T :` *<base class name>*	The type argument must be or derive from the specified base class. In a nullable context, `T` must be a non-nullable reference type derived from the specified base class.
`where T :` *<base class name>?*	The type argument must be or derive from the specified base class. In a nullable context, `T` can be either a nullable or non-nullable type derived from the specified base class.

Constraint	Description
`where T :` *<interface name>*	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context, `T` must be a non-nullable type that implements the specified interface.
`where T :` *<interface name>?*	The type argument must be or implement the specified interface. Multiple interface constraints can be specified. The constraining interface can also be generic. In a nullable context, `T` can be a nullable reference type, a non-nullable reference type, or a value type. `T` can't be a nullable value type.
`where T : U`	The type argument supplied for `T` must be or derive from the argument supplied for `U`. In a nullable context, if `U` is a non-nullable reference type, `T` must be a non-nullable reference type. If `U` is a nullable reference type, `T` can be either nullable or non-nullable.
`where T : default`	This constraint resolves the ambiguity when you need to specify an unconstrained type parameter when you override a method or provide an explicit interface implementation. The `default` constraint implies the base method without either the `class` or `struct` constraint. For more information, see the default constraint spec proposal.
`where T : allows ref struct`	This anti-constraint declares that the type argument for `T` can be a `ref struct` type. The generic type or method must obey ref safety rules for any instance of `T` because it might be a `ref struct`.

Some constraints are mutually exclusive, and some constraints must be in a specified order:

- You can apply at most one of the `struct`, `class`, `class?`, `notnull`, and `unmanaged` constraints. If you supply any of these constraints, it must be the first constraint specified for that type parameter.
- The base class constraint (`where T : Base` or `where T : Base?`) can't be combined with any of the constraints `struct`, `class`, `class?`, `notnull`, or `unmanaged`.
- You can apply at most one base class constraint, in either form. If you want to support the nullable base type, use `Base?`.
- You can't name both the non-nullable and nullable form of an interface as a constraint.
- The `new()` constraint can't be combined with the `struct` or `unmanaged` constraint. If you specify the `new()` constraint, it must be the last constraint for that type parameter. Anti-constraints, if applicable, can follow the `new()` constraint.
- The `default` constraint can be applied only on override or explicit interface implementations. It can't be combined with either the `struct` or `class` constraints.
- The `allows ref struct` anti-constraint can't be combined with the `class` or `class?` constraint.
- The `allows ref struct` anti-constraint must follow all constraints for that type parameter.

Why use constraints

Constraints specify the capabilities and expectations of a type parameter. Declaring those constraints means you can use the operations and method calls of the constraining type. You apply constraints to the type parameter when your generic class or method uses any operation on the generic members beyond simple assignment, which includes calling any methods not supported by System.Object. For example, the base class constraint tells the compiler that only objects of this type or derived from this type can replace that type argument. Once the compiler has this guarantee, it can allow methods of that type to be called in the generic class. The following code example demonstrates the functionality you can add to the `GenericList<T>` class (in Introduction to Generics) by applying a base class constraint.

C#

```
public class Employee
{
    public Employee(string name, int id) => (Name, ID) = (name, id);
    public string Name { get; set; }
    public int ID { get; set; }
}

public class GenericList<T> where T : Employee
{
    private class Node
    {
        public Node(T t) => (Next, Data) = (null, t);

        public Node? Next { get; set; }
        public T Data { get; set; }
    }

    private Node? head;

    public void AddHead(T t)
    {
        Node n = new(t) { Next = head };
        head = n;
    }

    public IEnumerator<T> GetEnumerator()
    {
        Node? current = head;

        while (current is not null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    public T? FindFirstOccurrence(string s)
    {
        Node? current = head;
```

```
        while (current is not null)
        {
            //The constraint enables access to the Name property.
            if (current.Data.Name == s)
            {
                return current.Data;
            }
            else
            {
                current = current.Next;
            }
        }
        return null;
    }
}
```

The constraint enables the generic class to use the `Employee.Name` property. The constraint specifies that all items of type `T` are guaranteed to be either an `Employee` object or an object that inherits from `Employee`.

Multiple constraints can be applied to the same type parameter, and the constraints themselves can be generic types, as follows:

C#

```
class EmployeeList<T> where T : notnull, Employee, IComparable<T>, new()
{
    public void AddDefault()
    {
        T t = new();
    }
}
```

When applying the `where T : class` constraint, avoid the `==` and `!=` operators on the type parameter because these operators test for reference identity only, not for value equality. This behavior occurs even if these operators are overloaded in a type that is used as an argument. The following code illustrates this point; the output is false even though the String class overloads the `==` operator.

C#

```
public static void OpEqualsTest<T>(T s, T t) where T : class
{
    Console.WriteLine(s == t);
}

private static void TestStringEquality()
{
```

```
    string s1 = "target";
    System.Text.StringBuilder sb = new("target");
    string s2 = sb.ToString();
    OpEqualsTest(s1, s2);
}
```

The compiler only knows that `T` is a reference type at compile time and must use the default operators that are valid for all reference types. If you must test for value equality, apply the `where T : IEquatable<T>` or `where T : IComparable<T>` constraint and implement the interface in any class used to construct the generic class.

Constraining multiple parameters

You can apply constraints to multiple parameters, and multiple constraints to a single parameter, as shown in the following example:

C#

```
class Base { }
class Test<T, U>
    where U : struct
    where T : Base, new()
{ }
```

Unbounded type parameters

Type parameters that have no constraints, such as T in public class `SampleClass<T>{}`, are called unbounded type parameters. Unbounded type parameters have the following rules:

- The `!=` and `==` operators can't be used because there's no guarantee that the concrete type argument supports these operators.
- They can be converted to and from `System.Object` or explicitly converted to any interface type.
- You can compare them to null. If an unbounded parameter is compared to `null`, the comparison always returns false if the type argument is a value type.

Type parameters as constraints

The use of a generic type parameter as a constraint is useful when a member function with its own type parameter has to constrain that parameter to the type parameter of the containing type, as shown in the following example:

C#

```
public class List<T>
{
    public void Add<U>(List<U> items) where U : T {/*...*/}
}
```

In the previous example, `T` is a type constraint in the context of the `Add` method, and an unbounded type parameter in the context of the `List` class.

Type parameters can also be used as constraints in generic class definitions. The type parameter must be declared within the angle brackets together with any other type parameters:

C#

```
//Type parameter V is used as a type constraint.
public class SampleClass<T, U, V> where T : V { }
```

The usefulness of type parameters as constraints with generic classes is limited because the compiler can assume nothing about the type parameter except that it derives from `System.Object`. Use type parameters as constraints on generic classes in scenarios in which you want to enforce an inheritance relationship between two type parameters.

`notnull` constraint

You can use the `notnull` constraint to specify that the type argument must be a non-nullable value type or non-nullable reference type. Unlike most other constraints, if a type argument violates the `notnull` constraint, the compiler generates a warning instead of an error.

The `notnull` constraint has an effect only when used in a nullable context. If you add the `notnull` constraint in a nullable oblivious context, the compiler doesn't generate any warnings or errors for violations of the constraint.

`class` constraint

The `class` constraint in a nullable context specifies that the type argument must be a non-nullable reference type. In a nullable context, when a type argument is a nullable reference type, the compiler generates a warning.

`default` constraint

The addition of nullable reference types complicates the use of `T?` in a generic type or method. `T?` can be used with either the `struct` or `class` constraint, but one of them must be present. When the `class` constraint was used, `T?` referred to the nullable reference type for `T`. `T?` can be used when neither constraint is applied. In that case, `T?` is interpreted as `T?` for value types and reference types. However, if `T` is an instance of Nullable<T>, `T?` is the same as `T`. In other words, it doesn't become `T??`.

Because `T?` can now be used without either the `class` or `struct` constraint, ambiguities can arise in overrides or explicit interface implementations. In both those cases, the override doesn't include the constraints, but inherits them from the base class. When the base class doesn't apply either the `class` or `struct` constraint, derived classes need to somehow specify an override applies to the base method without either constraint. The derived method applies the `default` constraint. The `default` constraint clarifies *neither* the `class` nor `struct` constraint.

Unmanaged constraint

You can use the `unmanaged` constraint to specify that the type parameter must be a non-nullable unmanaged type. The `unmanaged` constraint enables you to write reusable routines to work with types that can be manipulated as blocks of memory, as shown in the following example:

C#

```
unsafe public static byte[] ToByteArray<T>(this T argument) where T : unmanaged
{
    var size = sizeof(T);
    var result = new byte[size];
    byte* p = (byte*)&argument;
    for (var i = 0; i < size; i++)
        result[i] = *p++;
    return result;
}
```

The preceding method must be compiled in an `unsafe` context because it uses the `sizeof` operator on a type not known to be a built-in type. Without the `unmanaged` constraint, the `sizeof` operator is unavailable.

The `unmanaged` constraint implies the `struct` constraint and can't be combined with it. Because the `struct` constraint implies the `new()` constraint, the `unmanaged` constraint can't be combined with the `new()` constraint as well.

Delegate constraints

You can use System.Delegate or System.MulticastDelegate as a base class constraint. The Common Language Runtime (CLR) always allowed this constraint, but the C# language disallowed it. The `System.Delegate` constraint enables you to write code that works with delegates in a type-safe manner. The following code defines an extension method that combines two delegates provided they're the same type:

C#

```
extension<TDelegate>(TDelegate source) where TDelegate : System.Delegate
{
    public TDelegate? TypeSafeCombine(TDelegate target)
        => Delegate.Combine(source, target) as TDelegate;
}
```

You can use the preceding method to combine delegates that are the same type:

C#

```
Action first = () => Console.WriteLine("this");
Action second = () => Console.WriteLine("that");

var combined = first.TypeSafeCombine(second);
combined!();

Func<bool> test = () => true;
// Combine signature ensures combined delegates must
// have the same type.
//var badCombined = first.TypeSafeCombine(test);
```

If you uncomment the last line, it doesn't compile. Both `first` and `test` are delegate types, but they're different delegate types.

Enum constraints

You can also specify the System.Enum type as a base class constraint. The CLR always allowed this constraint, but the C# language disallowed it. Generics using `System.Enum` provide type-safe programming to cache results from using the static methods in `System.Enum`. The following sample finds all the valid values for an enum type, and then builds a dictionary that maps those values to its string representation.

C#

```
extension<T>(T) where T : System.Enum
{
    public static Dictionary<int, string> EnumNamedValues()
    {
        var result = new Dictionary<int, string>();
        var values = Enum.GetValues(typeof(T));

        foreach (int item in values)
            result.Add(item, Enum.GetName(typeof(T), item)!);
        return result;
    }
}
```

`Enum.GetValues` and `Enum.GetName` use reflection, which has performance implications. You can call `EnumNamedValues` to build a collection that is cached and reused rather than repeating the calls that require reflection.

You could use it as shown in the following sample to create an enum and build a dictionary of its values and names:

C#

```
enum Rainbow
{
    Red,
    Orange,
    Yellow,
    Green,
    Blue,
    Indigo,
    Violet
}
```

C#

```
var map = EnumNamedValues<Rainbow>();

foreach (var pair in map)
    Console.WriteLine($"{pair.Key}:\t{pair.Value}");
```

Type arguments implement declared interface

Some scenarios require that an argument supplied for a type parameter implement that interface. For example:

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract T operator +(T left, T right);
    static abstract T operator -(T left, T right);
}
```

This pattern enables the C# compiler to determine the containing type for the overloaded operators, or any `static virtual` or `static abstract` method. It provides the syntax so that the addition and subtraction operators can be defined on a containing type. Without this constraint, the parameters and arguments would be required to be declared as the interface, rather than the type parameter:

C#

```
public interface IAdditionSubtraction<T> where T : IAdditionSubtraction<T>
{
    static abstract IAdditionSubtraction<T> operator +(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);

    static abstract IAdditionSubtraction<T> operator -(
        IAdditionSubtraction<T> left,
        IAdditionSubtraction<T> right);
}
```

The preceding syntax would require implementers to use explicit interface implementation for those methods. Providing the extra constraint enables the interface to define the operators in terms of the type parameters. Types that implement the interface can implicitly implement the interface methods.

Allows ref struct

The `allows ref struct` anti-constraint declares that the corresponding type argument can be a ref struct type. Instances of that type parameter must obey the following rules:

- It can't be boxed.
- It participates in ref safety rules.
- Instances can't be used where a `ref struct` type isn't allowed, such as `static` fields.
- Instances can be marked with the `scoped` modifier.

The `allows ref struct` clause isn't inherited. In the following code:

C#

```
class SomeClass<T, S>
    where T : allows ref struct
    where S : T
{
    // etc
}
```

The argument for `S` can't be a `ref struct` because `S` doesn't have the `allows ref struct` clause.

A type parameter that has the `allows ref struct` clause can't be used as a type argument unless the corresponding type parameter also has the `allows ref struct` clause. This rule is demonstrated in the following example:

C#

```
public class Allow<T> where T : allows ref struct
{

}

public class Disallow<T>
{
}

public class Example<T> where T : allows ref struct
{
    private Allow<T> fieldOne; // Allowed. T is allowed to be a ref struct

    private Disallow<T> fieldTwo; // Error. T is not allowed to be a ref struct
}
```

The preceding sample shows that a type argument that might be a `ref struct` type can't be substituted for a type parameter that can't be a `ref struct` type.

See also

- System.Collections.Generic
- Introduction to Generics
- Generic Classes
- new Constraint

Last updated on 11/25/2025

Generic Classes (C# Programming Guide)

Article • 07/09/2022

Generic classes encapsulate operations that are not specific to a particular data type. The most common use for generic classes is with collections like linked lists, hash tables, stacks, queues, trees, and so on. Operations such as adding and removing items from the collection are performed in basically the same way regardless of the type of data being stored.

For most scenarios that require collection classes, the recommended approach is to use the ones provided in the .NET class library. For more information about using these classes, see Generic Collections in .NET.

Typically, you create generic classes by starting with an existing concrete class, and changing types into type parameters one at a time until you reach the optimal balance of generalization and usability. When creating your own generic classes, important considerations include the following:

- Which types to generalize into type parameters.

  As a rule, the more types you can parameterize, the more flexible and reusable your code becomes. However, too much generalization can create code that is difficult for other developers to read or understand.

- What constraints, if any, to apply to the type parameters (See Constraints on Type Parameters).

  A good rule is to apply the maximum constraints possible that will still let you handle the types you must handle. For example, if you know that your generic class is intended for use only with reference types, apply the class constraint. That will prevent unintended use of your class with value types, and will enable you to use the `as` operator on `T`, and check for null values.

- Whether to factor generic behavior into base classes and subclasses.

  Because generic classes can serve as base classes, the same design considerations apply here as with non-generic classes. See the rules about inheriting from generic base classes later in this topic.

- Whether to implement one or more generic interfaces.

> For example, if you are designing a class that will be used to create items in a generics-based collection, you may have to implement an interface such as IComparable<T> where `T` is the type of your class.

For an example of a simple generic class, see Introduction to Generics.

The rules for type parameters and constraints have several implications for generic class behavior, especially regarding inheritance and member accessibility. Before proceeding, you should understand some terms. For a generic class `Node<T>`, client code can reference the class either by specifying a type argument - to create a closed constructed type (`Node<int>`); or by leaving the type parameter unspecified - for example when you specify a generic base class, to create an open constructed type (`Node<T>`). Generic classes can inherit from concrete, closed constructed, or open constructed base classes:

C#

```
class BaseNode { }
class BaseNodeGeneric<T> { }

// concrete type
class NodeConcrete<T> : BaseNode { }

//closed constructed type
class NodeClosed<T> : BaseNodeGeneric<int> { }

//open constructed type
class NodeOpen<T> : BaseNodeGeneric<T> { }
```

Non-generic, in other words, concrete, classes can inherit from closed constructed base classes, but not from open constructed classes or from type parameters because there is no way at run time for client code to supply the type argument required to instantiate the base class.

C#

```
//No error
class Node1 : BaseNodeGeneric<int> { }

//Generates an error
//class Node2 : BaseNodeGeneric<T> {}

//Generates an error
//class Node3 : T {}
```

Generic classes that inherit from open constructed types must supply type arguments for any base class type parameters that are not shared by the inheriting class, as

demonstrated in the following code:

```
C#
```

```
class BaseNodeMultiple<T, U> { }

//No error
class Node4<T> : BaseNodeMultiple<T, int> { }

//No error
class Node5<T, U> : BaseNodeMultiple<T, U> { }

//Generates an error
//class Node6<T> : BaseNodeMultiple<T, U> {}
```

Generic classes that inherit from open constructed types must specify constraints that are a superset of, or imply, the constraints on the base type:

```
C#
```

```
class NodeItem<T> where T : System.IComparable<T>, new() { }
class SpecialNodeItem<T> : NodeItem<T> where T : System.IComparable<T>,
new() { }
```

Generic types can use multiple type parameters and constraints, as follows:

```
C#
```

```
class SuperKeyType<K, V, U>
    where U : System.IComparable<U>
    where V : new()
{ }
```

Open constructed and closed constructed types can be used as method parameters:

```
C#
```

```
void Swap<T>(List<T> list1, List<T> list2)
{
    //code to swap items
}

void Swap(List<int> list1, List<int> list2)
{
    //code to swap items
}
```

If a generic class implements an interface, all instances of that class can be cast to that interface.

Generic classes are invariant. In other words, if an input parameter specifies a `List<BaseClass>`, you will get a compile-time error if you try to provide a `List<DerivedClass>`.

See also

- System.Collections.Generic
- Generics
- Saving the State of Enumerators
- An Inheritance Puzzle, Part One

Generic Interfaces (C# Programming Guide)

It's often useful to define interfaces either for generic collection classes, or for the generic classes that represent items in the collection. To avoid boxing and unboxing operations on value types, it's better to use generic interfaces, such as IComparable<T>, on generic classes. The .NET class library defines several generic interfaces for use with the collection classes in the System.Collections.Generic namespace. For more information about these interfaces, see Generic interfaces.

When an interface is specified as a constraint on a type parameter, only types that implement the interface can be used. The following code example shows a `SortedList<T>` class that derives from the `GenericList<T>` class. For more information, see Introduction to Generics. `SortedList<T>` adds the constraint `where T : IComparable<T>`. This constraint enables the `BubbleSort` method in `SortedList<T>` to use the generic CompareTo method on list elements. In this example, list elements are a simple class, `Person` that implements `IComparable<Person>`.

C#

```
//Type parameter T in angle brackets.
public class GenericList<T> : System.Collections.Generic.IEnumerable<T>
{
    protected Node head;
    protected Node current = null;

    // Nested class is also generic on T
    protected class Node
    {
        public Node next;
        private T data;  //T as private member datatype

        public Node(T t)  //T used in non-generic constructor
        {
            next = null;
            data = t;
        }

        public Node Next
        {
            get { return next; }
            set { next = value; }
        }

        public T Data  //T as return type of property
        {
            get { return data; }
            set { data = value; }
        }
```

```
    }

    public GenericList()  //constructor
    {
        head = null;
    }

    public void AddHead(T t)  //T as method parameter type
    {
        Node n = new Node(t);
        n.Next = head;
        head = n;
    }

    // Implementation of the iterator
    public System.Collections.Generic.IEnumerator<T> GetEnumerator()
    {
        Node current = head;
        while (current != null)
        {
            yield return current.Data;
            current = current.Next;
        }
    }

    // IEnumerable<T> inherits from IEnumerable, therefore this class
    // must implement both the generic and non-generic versions of
    // GetEnumerator. In most cases, the non-generic method can
    // simply call the generic method.
    System.Collections.IEnumerator System.Collections.IEnumerable.GetEnumerator()
    {
        return GetEnumerator();
    }
}

public class SortedList<T> : GenericList<T> where T : System.IComparable<T>
{
    // A simple, unoptimized sort algorithm that
    // orders list elements from lowest to highest:

    public void BubbleSort()
    {
        if (null == head || null == head.Next)
        {
            return;
        }
        bool swapped;

        do
        {
            Node previous = null;
            Node current = head;
            swapped = false;

            while (current.next != null)
```

```
            {
                //  Because we need to call this method, the SortedList
                //  class is constrained on IComparable<T>
                if (current.Data.CompareTo(current.next.Data) > 0)
                {
                    Node tmp = current.next;
                    current.next = current.next.next;
                    tmp.next = current;

                    if (previous == null)
                    {
                        head = tmp;
                    }
                    else
                    {
                        previous.next = tmp;
                    }
                    previous = tmp;
                    swapped = true;
                }
                else
                {
                    previous = current;
                    current = current.next;
                }
            }
        } while (swapped);
    }
}

// A simple class that implements IComparable<T> using itself as the
// type argument. This is a common design pattern in objects that
// are stored in generic lists.
public class Person : System.IComparable<Person>
{
    string name;
    int age;

    public Person(string s, int i)
    {
        name = s;
        age = i;
    }

    // This will cause list elements to be sorted on age values.
    public int CompareTo(Person p)
    {
        return age - p.age;
    }

    public override string ToString()
    {
        return name + ":" + age;
    }
```

```
    // Must implement Equals.
    public bool Equals(Person p)
    {
        return (this.age == p.age);
    }
}

public class Program
{
    public static void Main()
    {
        //Declare and instantiate a new generic SortedList class.
        //Person is the type argument.
        SortedList<Person> list = new SortedList<Person>();

        //Create name and age values to initialize Person objects.
        string[] names =
        [
            "Franscoise",
            "Bill",
            "Li",
            "Sandra",
            "Gunnar",
            "Alok",
            "Hiroyuki",
            "Maria",
            "Alessandro",
            "Raul"
        ];

        int[] ages = [45, 19, 28, 23, 18, 9, 108, 72, 30, 35];

        //Populate the list.
        for (int x = 0; x < 10; x++)
        {
            list.AddHead(new Person(names[x], ages[x]));
        }

        //Print out unsorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with unsorted list");

        //Sort the list.
        list.BubbleSort();

        //Print out sorted list.
        foreach (Person p in list)
        {
            System.Console.WriteLine(p.ToString());
        }
        System.Console.WriteLine("Done with sorted list");
```

```
    }
}
```

Multiple interfaces can be specified as constraints on a single type, as follows:

C#

```
class Stack<T> where T : System.IComparable<T>, IEnumerable<T>
{
}
```

An interface can define more than one type parameter, as follows:

C#

```
interface IDictionary<K, V>
{
}
```

The rules of inheritance that apply to classes also apply to interfaces:

C#

```
interface IMonth<T> { }

interface IJanuary : IMonth<int> { }  //No error
interface IFebruary<T> : IMonth<int> { }  //No error
interface IMarch<T> : IMonth<T> { }    //No error
                                       //interface IApril<T>  : IMonth<T, U> {}
//Error
```

Generic interfaces can inherit from non-generic interfaces. In the .NET class library, IEnumerable<T> inherits from IEnumerable. When a generic interface inherits from a non-generic interface, the type parameter typically replaces `object` in the overridden members. For example, IEnumerable<T> uses `T` in place of `object` in the return value of GetEnumerator and in the Current property getter. Because `T` is used only in output positions in these members, IEnumerable<T> can be marked as covariant. If `T` were used in an input position in an overridden member, the interface couldn't be covariant, and the compiler would generate an error.

Concrete classes can implement closed constructed interfaces, as follows:

C#

```
interface IBaseInterface<T> { }
```

```
class SampleClass : IBaseInterface<string> { }
```

Generic classes can implement generic interfaces or closed constructed interfaces as long as the class parameter list supplies all arguments required by the interface, as follows:

C#

```
interface IBaseInterface1<T> { }
interface IBaseInterface2<T, U> { }

class SampleClass1<T> : IBaseInterface1<T> { }          //No error
class SampleClass2<T> : IBaseInterface2<T, string> { }  //No error
```

The rules that control method overloading are the same for methods within generic classes, generic structs, or generic interfaces. For more information, see Generic Methods.

Interfaces can declare `static abstract` or `static virtual` members. Interfaces that declare either `static abstract` or `static virtual` members are almost always generic interfaces. The compiler must resolve calls to `static virtual` and `static abstract` methods at compile time. `static virtual` and `static abstract` methods declared in interfaces don't have a runtime dispatch mechanism analogous to `virtual` or `abstract` methods declared in classes. Instead, the compiler uses type information available at compile time. These members are typically declared in generic interfaces. Furthermore, most interfaces that declare `static virtual` or `static abstract` methods declare that one of the type parameters must implement the declared interface. The compiler then uses the supplied type arguments to resolve the type of the declared member.

See also

- Introduction to Generics
- interface
- Generics

Last updated on 11/18/2025

Generic methods (C# programming guide)

Article • 03/27/2024

A generic method is a method that is declared with type parameters, as follows:

```C#
static void Swap<T>(ref T lhs, ref T rhs)
{
    T temp;
    temp = lhs;
    lhs = rhs;
    rhs = temp;
}
```

The following code example shows one way to call the method by using `int` for the type argument:

```C#
public static void TestSwap()
{
    int a = 1;
    int b = 2;

    Swap<int>(ref a, ref b);
    System.Console.WriteLine(a + " " + b);
}
```

You can also omit the type argument and the compiler will infer it. The following call to `Swap` is equivalent to the previous call:

```C#
Swap(ref a, ref b);
```

The same rules for type inference apply to static methods and instance methods. The compiler can infer the type parameters based on the method arguments you pass in; it cannot infer the type parameters only from a constraint or return value. Therefore type inference does not work with methods that have no parameters. Type inference occurs at compile time before the compiler tries to resolve overloaded method signatures. The compiler applies type inference logic to all generic methods that share the same name.

In the overload resolution step, the compiler includes only those generic methods on which type inference succeeded.

Within a generic class, non-generic methods can access the class-level type parameters, as follows:

```
C#
```

```
class SampleClass<T>
{
    void Swap(ref T lhs, ref T rhs) { }
}
```

If you define a generic method that takes the same type parameters as the containing class, the compiler generates warning CS0693 because within the method scope, the argument supplied for the inner `T` hides the argument supplied for the outer `T`. If you require the flexibility of calling a generic class method with type arguments other than the ones provided when the class was instantiated, consider providing another identifier for the type parameter of the method, as shown in `GenericList2<T>` in the following example.

```
C#
```

```
class GenericList<T>
{
    // CS0693.
    void SampleMethod<T>() { }
}

class GenericList2<T>
{
    // No warning.
    void SampleMethod<U>() { }
}
```

Use constraints to enable more specialized operations on type parameters in methods. This version of `Swap<T>`, now named `SwapIfGreater<T>`, can only be used with type arguments that implement IComparable<T>.

```
C#
```

```
void SwapIfGreater<T>(ref T lhs, ref T rhs) where T : System.IComparable<T>
{
    T temp;
    if (lhs.CompareTo(rhs) > 0)
    {
        temp = lhs;
```

```
            lhs = rhs;
            rhs = temp;
        }
    }
```

Generic methods can be overloaded on several type parameters. For example, the following methods can all be located in the same class:

C#

```
void DoWork() { }
void DoWork<T>() { }
void DoWork<T, U>() { }
```

You can also use the type parameter as the return type of a method. The following code example shows a method that returns an array of type `T`:

C#

```
T[] Swap<T>(T a, T b)
{
    return [b, a];
}
```

C# Language Specification

For more information, see the C# Language Specification.

See also

- System.Collections.Generic
- Introduction to Generics
- Methods

Generics and Arrays (C# Programming Guide)

Article • 03/12/2024

Single-dimensional arrays that have a lower bound of zero automatically implement IList<T>. This enables you to create generic methods that can use the same code to iterate through arrays and other collection types. This technique is primarily useful for reading data in collections. The IList<T> interface cannot be used to add or remove elements from an array. An exception will be thrown if you try to call an IList<T> method such as RemoveAt on an array in this context.

The following code example demonstrates how a single generic method that takes an IList<T> input parameter can iterate through both a list and an array, in this case an array of integers.

C#

```
class Program
{
    static void Main()
    {
        int[] arr = [0, 1, 2, 3, 4];
        List<int> list = new List<int>();

        for (int x = 5; x < 10; x++)
        {
            list.Add(x);
        }

        ProcessItems<int>(arr);
        ProcessItems<int>(list);
    }

    static void ProcessItems<T>(IList<T> coll)
    {
        // IsReadOnly returns True for the array and False for the List.
        System.Console.WriteLine
            ("IsReadOnly returns {0} for this collection.",
            coll.IsReadOnly);

        // The following statement causes a run-time exception for the
        // array, but not for the List.
        //coll.RemoveAt(4);

        foreach (T item in coll)
        {
            System.Console.Write(item?.ToString() + " ");
        }
```

```
            System.Console.WriteLine();
        }
    }
```

See also

- System.Collections.Generic
- Generics
- Arrays
- Generics

Generic Delegates (C# Programming Guide)

Article • 03/12/2024

A delegate can define its own type parameters. Code that references the generic delegate can specify the type argument to create a closed constructed type, just like when instantiating a generic class or calling a generic method, as shown in the following example:

```
C#
```

```
public delegate void Del<T>(T item);
public static void Notify(int i) { }

Del<int> m1 = new Del<int>(Notify);
```

C# version 2.0 has a new feature called method group conversion, which applies to concrete as well as generic delegate types, and enables you to write the previous line with this simplified syntax:

```
C#
```

```
Del<int> m2 = Notify;
```

Delegates defined within a generic class can use the generic class type parameters in the same way that class methods do.

```
C#
```

```
class Stack<T>
{
    public delegate void StackDelegate(T[] items);
}
```

Code that references the delegate must specify the type argument of the containing class, as follows:

```
C#
```

```
private static void DoWork(float[] items) { }

public static void TestStack()
{
```

```
    Stack<float> s = new Stack<float>();
    Stack<float>.StackDelegate d = DoWork;
}
```

Generic delegates are especially useful in defining events based on the typical design pattern because the sender argument can be strongly typed and no longer has to be cast to and from Object.

C#

```
delegate void StackEventHandler<T, U>(T sender, U eventArgs);

class Stack<T>
{
    public class StackEventArgs : System.EventArgs { }
    public event StackEventHandler<Stack<T>, StackEventArgs>? StackEvent;

    protected virtual void OnStackChanged(StackEventArgs a)
    {
        if (StackEvent is not null)
            StackEvent(this, a);
    }
}

class SampleClass
{
    public void HandleStackChange<T>(Stack<T> stack, Stack<T>.StackEventArgs
args) { }
}

public static void Test()
{
    Stack<double> s = new Stack<double>();
    SampleClass o = new SampleClass();
    s.StackEvent += o.HandleStackChange;
}
```

See also

- System.Collections.Generic
- Introduction to Generics
- Generic Methods
- Generic Classes
- Generic Interfaces
- Delegates
- Generics

Differences Between C++ Templates and C# Generics (C# Programming Guide)

Article • 03/12/2024

C# Generics and C++ templates are both language features that provide support for parameterized types. However, there are many differences between the two. At the syntax level, C# generics are a simpler approach to parameterized types without the complexity of C++ templates. In addition, C# does not attempt to provide all of the functionality that C++ templates provide. At the implementation level, the primary difference is that C# generic type substitutions are performed at run time and generic type information is thereby preserved for instantiated objects. For more information, see Generics in the Runtime.

The following are the key differences between C# Generics and C++ templates:

- C# generics do not provide the same amount of flexibility as C++ templates. For example, it is not possible to call arithmetic operators in a C# generic class, although it is possible to call user defined operators.
- C# does not allow non-type template parameters, such as `template C<int i> {}`.
- C# does not support explicit specialization; that is, a custom implementation of a template for a specific type.
- C# does not support partial specialization: a custom implementation for a subset of the type arguments.
- C# does not allow the type parameter to be used as the base class for the generic type.
- C# does not allow type parameters to have default types.
- In C#, a generic type parameter cannot itself be a generic, although constructed types can be used as generics. C++ does allow template parameters.
- C++ allows code that might not be valid for all type parameters in the template, which is then checked for the specific type used as the type parameter. C# requires code in a class to be written in such a way that it will work with any type that satisfies the constraints. For example, in C++ it is possible to write a function that uses the arithmetic operators `+` and `-` on objects of the type parameter, which will

produce an error at the time of instantiation of the template with a type that does not support these operators. C# disallows this; the only language constructs allowed are those that can be deduced from the constraints.

See also

- Introduction to Generics
- Templates

Generics in the runtime (C# programming guide)

Article • 04/20/2024

When a generic type or method is compiled into common intermediate language (CIL), it contains metadata that identifies it as having type parameters. How the CIL for a generic type is used differs based on whether the supplied type parameter is a value type or reference type.

When a generic type is first constructed with a value type as a parameter, the runtime creates a specialized generic type with the supplied parameter or parameters substituted in the appropriate locations in the CIL. Specialized generic types are created one time for each unique value type that is used as a parameter.

For example, suppose your program code declared a stack that's constructed of integers:

```
C#

Stack<int>? stack;
```

At this point, the runtime generates a specialized version of the Stack<T> class that has the integer substituted appropriately for its parameter. Now, whenever your program code uses a stack of integers, the runtime reuses the generated specialized Stack<T> class. In the following example, two instances of a stack of integers are created, and they share a single instance of the `Stack<int>` code:

```
C#

Stack<int> stackOne = new Stack<int>();
Stack<int> stackTwo = new Stack<int>();
```

However, suppose that another Stack<T> class with a different value type such as a `long` or a user-defined structure as its parameter is created at another point in your code. As a result, the runtime generates another version of the generic type and substitutes a `long` in the appropriate locations in CIL. Conversions are no longer necessary because each specialized generic class natively contains the value type.

Generics work somewhat differently for reference types. The first time a generic type is constructed with any reference type, the runtime creates a specialized generic type with object references substituted for the parameters in the CIL. Then, every time that a

constructed type is instantiated with a reference type as its parameter, regardless of what type it is, the runtime reuses the previously created specialized version of the generic type. This is possible because all references are the same size.

For example, suppose you had two reference types, a `Customer` class and an `Order` class, and also suppose that you created a stack of `Customer` types:

C#

```
class Customer { }
class Order { }
```

C#

```
Stack<Customer> customers;
```

At this point, the runtime generates a specialized version of the Stack<T> class that stores object references that will be filled in later instead of storing data. Suppose the next line of code creates a stack of another reference type, which is named `Order`:

C#

```
Stack<Order> orders = new Stack<Order>();
```

Unlike with value types, another specialized version of the Stack<T> class is not created for the `Order` type. Instead, an instance of the specialized version of the Stack<T> class is created and the `orders` variable is set to reference it. Suppose that you then encountered a line of code to create a stack of a `Customer` type:

C#

```
customers = new Stack<Customer>();
```

As with the previous use of the Stack<T> class created by using the `Order` type, another instance of the specialized Stack<T> class is created. The pointers that are contained therein are set to reference an area of memory the size of a `Customer` type. Because the number of reference types can vary wildly from program to program, the C# implementation of generics greatly reduces the amount of code by reducing to one the number of specialized classes created by the compiler for generic classes of reference types.

Moreover, when a generic C# class is instantiated by using a value type or reference type parameter, reflection can query it at run time and both its actual type and its type parameter can be ascertained.

See also

- System.Collections.Generic
- Introduction to Generics
- Generics

C# language reference

The language reference provides an informal reference to C# syntax and idioms for beginners and experienced C# and .NET developers.

C# language reference

OVERVIEW

C# language strategy

REFERENCE

C# keywords

C# operators and expressions

Configure language version

C# language specification - C# 8 draft in progress

What's new

WHAT'S NEW

What's new in C# 14

What's new in C# 13

What's new in C# 12

REFERENCE

Breaking changes in the C# compiler

Version compatibility

Stay in touch

REFERENCE

.NET developer community

YouTube

Twitter

Specifications

Read the detailed specification for the C# language and the detailed specifications for the latest features.

ECMA specifications and latest features

OVERVIEW

Specification process

Detailed ECMA specification contents

REFERENCE

Latest feature specifications

Draft C# ECMA specification - Introductory material

REFERENCE

Foreword

Introduction

REFERENCE

Scope

Normative references

Terms and definitions

General description

Conformance

Draft C# ECMA specification - Language specification

REFERENCE

Lexical structure

Basic concepts

Types

Variables

Conversions

Patterns

Expressions

Statements

REFERENCE

Namespaces

Classes

Structs

Arrays

Interfaces

Enums

Delegates

Exceptions

Attributes

Unsafe code

Draft C# ECMA specification - Annexes

REFERENCE

Grammar

Portability issues

Standard library

Documentation comments

Bibliography