

What's new in C# 8.0

09/20/2019 • 15 minutes to read •  +11

In this article

[Readonly members](#)

[Default interface methods](#)

[More patterns in more places](#)

[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](#)

C# 8.0 adds the following features and enhancements to the C# language:

- [Readonly members](#)
- [Default interface methods](#)
- [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](#)

The remainder of this article briefly describes these features. Where in-depth articles are available, links to those tutorials and overviews are provided. You can explore these features in your environment using the `dotnet try` global tool:

1. Install the [dotnet-try](#) global tool.
2. Clone the [dotnet/try-samples](#) repository.
3. Set the current directory to the *csharp8* subdirectory for the *try-samples* repository.
4. Run `dotnet try`.

Readonly members

You can apply the `readonly` modifier to members of a struct. It indicates that the member doesn't modify state. It's more granular than applying the `readonly` modifier to a `struct` declaration. Consider the following mutable struct:

C#

 Copy

```
public struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Distance => Math.Sqrt(X * X + Y * Y);
}
```

```
public override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
}
```

Like most structs, the `ToString()` method doesn't modify state. You could indicate that by adding the `readonly` modifier to the declaration of `ToString()`:

C#

 Copy

```
public readonly override string ToString() =>
    $"({X}, {Y}) is {Distance} from the origin";
```

The preceding change generates a compiler warning, because `ToString` accesses the `Distance` property, which isn't marked `readonly`:

console

 Copy

```
warning CS8656: Call to non-readonly member 'Point.Distance.get' from a 'readonly' member results in an
implicit copy of 'this'
```

The compiler warns you when it needs to create a defensive copy. The `Distance` property doesn't change state, so you can fix this warning by adding the `readonly` modifier to the declaration:

C#

 Copy

```
public readonly double Distance => Math.Sqrt(X * X + Y * Y);
```

Notice that the `readonly` modifier is necessary on a read-only property. The compiler doesn't assume `get` accessors don't modify state; you must declare `readonly` explicitly. Auto-implemented properties are an exception; the compiler will treat all auto-implemented getters as `readonly`, so here there's no need to add the `readonly` modifier to the `x` and `y` properties.

The compiler does enforce the rule that `readonly` members don't modify state. The following method won't compile unless you remove the `readonly` modifier:

C#



```
public readonly void Translate(int xOffset, int yOffset)
{
    X += xOffset;
    Y += yOffset;
}
```

This feature lets you specify your design intent so the compiler can enforce it, and make optimizations based on that intent. You can learn more about `readonly` members in the language reference article on [readonly](#).

Default interface methods

You can now add members to interfaces and provide an implementation for those members. This language feature enables API authors to add methods to an interface in later versions without breaking source or binary compatibility with existing implementations of that interface. Existing implementations *inherit* the default implementation. This feature also enables C# to interoperate with APIs that target Android or Swift, which support similar features. Default interface methods also enable scenarios similar to a "traits" language feature.

Default interface methods affects many scenarios and language elements. Our first tutorial covers [updating an interface with default implementations](#). Other tutorials and reference updates are coming in time for general release.

More patterns in more places

Pattern matching gives tools to provide shape-dependent functionality across related but different kinds of data. C# 7.0 introduced syntax for type patterns and constant patterns by using the `is` expression and the `switch` statement. These features represented the first tentative steps toward supporting programming paradigms where data and functionality live


apart. As the industry moves toward more microservices and other cloud-based architectures, other language tools are needed.

C# 8.0 expands this vocabulary so you can use more pattern expressions in more places in your code. Consider these features when your data and functionality are separate. Consider pattern matching when your algorithms depend on a fact other than the runtime type of an object. These techniques provide another way to express designs.


In addition to new patterns in new places, C# 8.0 adds **recursive patterns**. The result of any pattern expression is an expression. A recursive pattern is simply a pattern expression applied to the output of another pattern expression.

Switch expressions

Often, a [switch](#) statement produces a value in each of its case blocks. **Switch expressions** enable you to use more concise expression syntax. There are fewer repetitive case and break keywords, and fewer curly braces. As an example, consider the following enum that lists the colors of the rainbow:

C#	
<pre>public enum Rainbow { Red, Orange, Yellow, Green, Blue, Indigo, Violet }</pre>	

If your application defined an `RGBColor` type that is constructed from the `R`, `G` and `B` components, you could convert a `Rainbow` value to its RGB values using the following method containing a switch expression:

	
--	---

C#

 Copy

```
public static RGBColor FromRainbow(Rainbow colorBand) =>
    colorBand switch
    {
        Rainbow.Red    => new RGBColor(0xFF, 0x00, 0x00),
        Rainbow.Orange => new RGBColor(0xFF, 0x7F, 0x00),
        Rainbow.Yellow => new RGBColor(0xFF, 0xFF, 0x00),
        Rainbow.Green  => new RGBColor(0x00, 0xFF, 0x00),
        Rainbow.Blue   => new RGBColor(0x00, 0x00, 0xFF),
        Rainbow.Indigo => new RGBColor(0x4B, 0x00, 0x82),
        Rainbow.Violet => new RGBColor(0x94, 0x00, 0xD3),
        _               => throw new ArgumentException(message: "invalid enum value", paramName:
nameof(colorBand)),
    };
```

There are several syntax improvements here:

- The variable comes before the `switch` keyword. The different order makes it visually easy to distinguish the switch expression from the switch statement.
- The `case` and `:` elements are replaced with `=>`. It's more concise and intuitive.
- The `default` case is replaced with a `_` discard.
- The bodies are expressions, not statements.

Contrast that with the equivalent code using the classic `switch` statement:

C#

 Copy

```
public static RGBColor FromRainbowClassic(Rainbow colorBand)
{
    switch (colorBand)
    {
        case Rainbow.Red:
            return new RGBColor(0xFF, 0x00, 0x00);
        case Rainbow.Orange:
```

```
        return new RGBColor(0xFF, 0x7F, 0x00);
    case Rainbow.Yellow:
        return new RGBColor(0xFF, 0xFF, 0x00);
    case Rainbow.Green:
        return new RGBColor(0x00, 0xFF, 0x00);
    case Rainbow.Blue:
        return new RGBColor(0x00, 0x00, 0xFF);
    case Rainbow.Indigo:
        return new RGBColor(0x4B, 0x00, 0x82);
    case Rainbow.Violet:
        return new RGBColor(0x94, 0x00, 0xD3);
    default:
        throw new ArgumentException(message: "invalid enum value", paramName: nameof(colorBand));
    };
}
```

Property patterns

The **property pattern** enables you to match on properties of the object examined. Consider an eCommerce site that must compute sales tax based on the buyer's address. That computation isn't a core responsibility of an `Address` class. It will change over time, likely more often than address format changes. The amount of sales tax depends on the `State` property of the address. The following method uses the property pattern to compute the sales tax from the address and the price:

C#



```
public static decimal ComputeSalesTax(Address location, decimal salePrice) =>
    location switch
    {
        { State: "WA" } => salePrice * 0.06M,
        { State: "MN" } => salePrice * 0.75M,
        { State: "MI" } => salePrice * 0.05M,
        // other cases removed for brevity...
        _ => 0M
    };
};
```

Pattern matching creates a concise syntax for expressing this algorithm.

Tuple patterns

Some algorithms depend on multiple inputs. **Tuple patterns** allow you to switch based on multiple values expressed as a [tuple](#). The following code shows a switch expression for the game *rock, paper, scissors*:

C#

 Copy

```
public static string RockPaperScissors(string first, string second)
=> (first, second) switch
{
    ("rock", "paper") => "rock is covered by paper. Paper wins.",
    ("rock", "scissors") => "rock breaks scissors. Rock wins.",
    ("paper", "rock") => "paper covers rock. Paper wins.",
    ("paper", "scissors") => "paper is cut by scissors. Scissors wins.",
    ("scissors", "rock") => "scissors is broken by rock. Rock wins.",
    ("scissors", "paper") => "scissors cuts paper. Scissors wins.",
    (_, _) => "tie"
};
```

The messages indicate the winner. The discard case represents the three combinations for ties, or other text inputs.

Positional patterns

Some types include a `Deconstruct` method that deconstructs its properties into discrete variables. When a `Deconstruct` method is accessible, you can use **positional patterns** to inspect properties of the object and use those properties for a pattern. Consider the following `Point` class that includes a `Deconstruct` method to create discrete variables for `x` and `y`:

C#

 Copy

```
public class Point
{
```



```
public int X { get; }
public int Y { get; }

public Point(int x, int y) => (X, Y) = (x, y);

public void Deconstruct(out int x, out int y) =>
    (x, y) = (X, Y);
}
```

Additionally, consider the following enum that represents various positions of a quadrant:

C#

 Copy

```
public enum Quadrant
{
    Unknown,
    Origin,
    One,
    Two,
    Three,
    Four,
    OnBorder
}
```

The following method uses the **positional pattern** to extract the values of `x` and `y`. Then, it uses a `when` clause to determine the Quadrant of the point:

C#

 Copy

```
static Quadrant GetQuadrant(Point point) => point switch
{
    (0, 0) => Quadrant.Origin,
    var (x, y) when x > 0 && y > 0 => Quadrant.One,
    var (x, y) when x < 0 && y > 0 => Quadrant.Two,
    var (x, y) when x < 0 && y < 0 => Quadrant.Three,
```

```
var (x, y) when x > 0 && y < 0 => Quadrant.Four,  
var (_, _) => Quadrant.OnBorder,  
_ => Quadrant.Unknown  
};
```

The discard pattern in the preceding switch matches when either `x` or `y` is 0, but not both. A switch expression must either produce a value or throw an exception. If none of the cases match, the switch expression throws an exception. The compiler generates a warning for you if you don't cover all possible cases in your switch expression.

You can explore pattern matching techniques in this [advanced tutorial on pattern matching](#).

Using declarations

A **using declaration** is a variable declaration preceded by the `using` keyword. It tells the compiler that the variable being declared should be disposed at the end of the enclosing scope. For example, consider the following code that writes a text file:

C#

 Copy

```
static int WriteLinesToFile(IEnumerable<string> lines)  
{  
    using var file = new System.IO.StreamWriter("WriteLines2.txt");  
    // Notice how we declare skippedLines after the using statement.  
    int skippedLines = 0;  
    foreach (string line in lines)  
    {  
        if (!line.Contains("Second"))  
        {  
            file.WriteLine(line);  
        }  
        else  
        {  
            skippedLines++;  
        }  
    }  
}
```

```
}  
// Notice how skippedLines is in scope here.  
return skippedLines;  
// file is disposed here  
}
```

In the preceding example, the file is disposed when the closing brace for the method is reached. That's the end of the scope in which `file` is declared. The preceding code is equivalent to the following code that uses the classic [using statement](#):

C#

 Copy

```
static int WriteLinesToFile(IEnumerable<string> lines)  
{  
    // We must declare the variable outside of the using block  
    // so that it is in scope to be returned.  
    int skippedLines = 0;  
    using (var file = new System.IO.StreamWriter("WriteLines2.txt"))  
    {  
        foreach (string line in lines)  
        {  
            if (!line.Contains("Second"))  
            {  
                file.WriteLine(line);  
            }  
            else  
            {  
                skippedLines++;  
            }  
        }  
    } // file is disposed here  
    return skippedLines;  
}
```

In the preceding example, the file is disposed when the closing brace associated with the `using` statement is reached.

In both cases, the compiler generates the call to `Dispose()`. The compiler generates an error if the expression in the `using` statement isn't disposable.

Static local functions

You can now add the `static` modifier to local functions to ensure that local function doesn't capture (reference) any variables from the enclosing scope. Doing so generates CS8421, "A static local function can't contain a reference to <variable>."

Consider the following code. The local function `LocalFunction` accesses the variable `y`, declared in the enclosing scope (the method `M`). Therefore, `LocalFunction` can't be declared with the `static` modifier:

C#

 Copy

```
int M()
{
    int y;
    LocalFunction();
    return y;

    void LocalFunction() => y = 0;
}
```

The following code contains a static local function. It can be static because it doesn't access any variables in the enclosing scope:

C#

 Copy

```
int M()
{
    int y = 5;
    int x = 7;
```

```
return Add(x, y);

static int Add(int left, int right) => left + right;
}
```

Disposable ref structs

A struct declared with the `ref` modifier may not implement any interfaces and so can't implement [IDisposable](#). Therefore, to enable a `ref` struct to be disposed, it must have an accessible `void Dispose()` method. This feature also applies to `readonly ref` struct declarations.

Nullable reference types

Inside a nullable annotation context, any variable of a reference type is considered to be a **nonnullable reference type**. If you want to indicate that a variable may be null, you must append the type name with the `?` to declare the variable as a **nullable reference type**.

For nonnullable reference types, the compiler uses flow analysis to ensure that local variables are initialized to a non-null value when declared. Fields must be initialized during construction. The compiler generates a warning if the variable isn't set by a call to any of the available constructors or by an initializer. Furthermore, nonnullable reference types can't be assigned a value that could be null.

Nullable reference types aren't checked to ensure they aren't assigned or initialized to null. However, the compiler uses flow analysis to ensure that any variable of a nullable reference type is checked against null before it's accessed or assigned to a nonnullable reference type.

You can learn more about the feature in the overview of [nullable reference types](#). Try it yourself in a new application in this [nullable reference types tutorial](#). Learn about the steps to migrate an existing codebase to make use of nullable reference types in the [migrating an application to use nullable reference types tutorial](#).

Asynchronous streams

Starting with C# 8.0, you can create and consume streams asynchronously. A method that returns an asynchronous stream has three properties:

1. It's declared with the `async` modifier.
2. It returns an [IAsyncEnumerable<T>](#).
3. The method contains `yield return` statements to return successive elements in the asynchronous stream.

Consuming an asynchronous stream requires you to add the `await` keyword before the `foreach` keyword when you enumerate the elements of the stream. Adding the `await` keyword requires the method that enumerates the asynchronous stream to be declared with the `async` modifier and to return a type allowed for an `async` method. Typically that means returning a [Task](#) or [Task<TResult>](#). It can also be a [ValueTask](#) or [ValueTask<TResult>](#). A method can both consume and produce an asynchronous stream, which means it would return an [IAsyncEnumerable<T>](#). The following code generates a sequence from 0 to 19, waiting 100 ms between generating each number:

C#



```
public static async System.Collections.Generic.IAsyncEnumerable<int> GenerateSequence()  
{  
    for (int i = 0; i < 20; i++)  
    {  
        await Task.Delay(100);  
        yield return i;  
    }  
}
```

You would enumerate the sequence using the `await foreach` statement:

C#



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

You can try asynchronous streams yourself in our tutorial on [creating and consuming async streams](#).

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 the 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 indexes. Consider an array `sequence`. The `0` index is the same as `sequence[0]`. The `^0` index is the same as `sequence[sequence.Length]`. Note that `sequence[^0]` does throw an exception, just as `sequence[sequence.Length]` does. For any number `n`, the index `^n` is the same as `sequence.Length - n`.

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.

Let's look at a few examples. Consider the following array, annotated with its index from the start and from the end:

C#



```
var words = new string[]
{
    "The",           // index from start    index from end
    "quick",         // 0                ^9
    "brown",         // 1                ^8
    "fox",           // 2                ^7
    "jumped",        // 3                ^6
    "over",          // 4                ^5
    "the",           // 5                ^4
    "lazy",          // 6                ^3
    "dog",           // 7                ^2
    "dog",           // 8                ^1
};                  // 9 (or words.Length) ^0
```

You can retrieve the last word with the `^1` index:

C#

 Copy

```
Console.WriteLine($"The last word is {words[^1]}");
// writes "dog"
```

The following code creates a subrange with the words "quick", "brown", and "fox". It includes `words[1]` through `words[3]`. The element `words[4]` isn't in the range.

C#

 Copy

```
var quickBrownFox = words[1..4];
```

The following code creates a subrange with "lazy" and "dog". It includes `words[^2]` and `words[^1]`. The end index `words[^0]` isn't included:

C#

 Copy


```
var lazyDog = words[^2..^0];
```

The following examples create ranges that are open ended for the start, end, or both:

C#

 Copy

```
var allWords = words[..]; // contains "The" through "dog".  
var firstPhrase = words[..4]; // contains "The" through "fox"  
var lastPhrase = words[6..]; // contains "the", "lazy" and "dog"
```

You can also declare ranges as variables:

C#

 Copy

```
Range phrase = 1..4;
```

The range can then be used inside the [and] characters:

C#

 Copy

```
var text = words[phrase];
```

Not only arrays support indices and ranges. You also can use indices and ranges with [string](#), [Span<T>](#), or [ReadOnlySpan<T>](#). For more information, see [Type support for indices and ranges](#).

You can explore more about indices and ranges in the tutorial on [indices and ranges](#).

Null-coalescing assignment

C# 8.0 introduces the null-coalescing assignment operator `??=`. You can use the `??=` operator to assign the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`.

C#

 Copy

```
List<int> numbers = null;
int? i = null;

numbers ??= new List<int>();
numbers.Add(i ??= 17);
numbers.Add(i ??= 20);

Console.WriteLine(string.Join(" ", numbers)); // output: 17 17
Console.WriteLine(i); // output: 17
```

For more information, see the [?? and ??= operators](#) article.

Unmanaged constructed types

In C# 7.3 and earlier, a constructed type (a type that includes at least one type argument) can't be an [unmanaged type](#). Starting with C# 8.0, a constructed value type is unmanaged if it contains fields of unmanaged types only.

For example, given the following definition of the generic `Coords<T>` type:

C#

 Copy

```
public struct Coords<T>
{
    public T X;
    public T Y;
}
```

the `Coords<int>` type is an unmanaged type in C# 8.0 and later. Like for any unmanaged type, you can create a pointer to a variable of this type or [allocate a block of memory on the stack](#) for instances of this type:

C#



```
Span<Coords<int>> coordinates = stackalloc[]
{
    new Coords<int> { X = 0, Y = 0 },
    new Coords<int> { X = 0, Y = 3 },
    new Coords<int> { X = 4, Y = 0 }
};
```

For more information, see [Unmanaged types](#).

Stackalloc in nested expressions

Starting with C# 8.0, if the result of a [stackalloc](#) expression is of the [System.Span<T>](#) or [System.ReadOnlySpan<T>](#) type, you can use the `stackalloc` expression in other expressions:

C#



```
Span<int> numbers = stackalloc[] { 1, 2, 3, 4, 5, 6 };
var ind = numbers.IndexOfAny(stackalloc[] { 2, 4, 6, 8 });
Console.WriteLine(ind); // output: 1
```

Enhancement of interpolated verbatim strings

Order of the `$` and `@` tokens in [interpolated](#) verbatim strings can be any: both `$@"..."` and `@$"..."` are valid interpolated verbatim strings. In earlier C# versions, the `$` token must appear before the `@` token.

