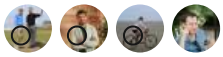
What's new in C# 10

11/17/20217 minutes to read 

Is this page helpful?

In this article

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
- Improvements on lambda expressions
- Allow const interpolated strings
- Record types can seal ToString()
- Improved definite assignment
- Allow both assignment and declaration in the same deconstruction
- Allow AsyncMethodBuilder attribute on methods
- CallerArgumentExpression attribute
- Enhanced #line pragma

Additional features are available in *preview* mode. You're encouraged to try these features and provide feedback on them. They may change before their final release. In order to use these features, you must set <LangVersion> to Preview in your project. Read about Generic attributes later in this article.

C# 10 is supported on **.NET 6**. For more information, see C# language versioning.

You can download the latest .NET 6 SDK from the .NET downloads page. You can also download Visual Studio 2022 preview, which includes the .NET 6 SDK.

Record structs

You can declare value type records using the record struct or readonly record structdeclarations. You can now clarify that a `record` is a reference type with the `record class` declaration.

Improvements of structure types

C# 10 introduces the following improvements related to structure types:

- You can declare an instance parameterless constructor in a structure type and initialize an instance field or property at its declaration. For more information, see the Parameterless constructors and field initializers section of the Structure typesarticle.
- A left-hand operand of the with expression can be of any structure type or an anonymous (reference) type.

Interpolated string handler

You can create a type that builds the resulting string from an interpolated string expression. The .NET libraries use this feature in many APIs. You can build one by following this tutorial.

Global using directives

You can add the `global` modifier to any using directive to instruct the compiler that the directive applies to all source files in the compilation. This is typically all source files in a project.

File-scoped namespace declaration

You can use a new form of the namespace declaration to declare that all declarations that follow are members of the declared namespace:

C# Copy

```
namespace MyNamespace;
```

This new syntax saves both horizontal and vertical space for the most common `namespace` declarations.

Extended property patterns

Beginning with C# 10, you can reference nested properties or fields within a property pattern. For example, a pattern of the form

C# Copy

```
{ Prop1.Prop2: pattern }
```

is valid in C# 10 and later and equivalent to

C# Copy

```
{ Prop1: { Prop2: pattern } }
```

valid in C# 8.0 and later.

For more information, see the Extended property patterns feature proposal note. For more information about a property pattern, see the Property pattern section of the Patterns article.

Lambda expression improvements

C# 10 includes many improvements to how lambda expressions are handled:

- Lambda expressions may have a natural type, where the compiler can infer a delegate type from the lambda expression or method group.
- Lambda expressions may declare a return type when the compiler can't infer it.
- Attributes can be applied to lambda expressions.

These features make lambda expressions more similar to methods and local functions. They make it easier to use lambda expressions without declaring a variable of a delegate type, and they work more seamlessly with the new ASP.NET Core Minimal APIs.

Constant interpolated strings

In C# 10, `const` strings may be initialized using string interpolation if all the placeholders are themselves constant strings. String interpolation can create more readable constant strings as you build constant strings used in your application. The placeholder expressions can't be numeric constants because those constants are converted to strings at run time. The current culture may affect their string representation. Learn more in the language reference on const expressions.

Record types can seal ToString

In C# 10, you can add the `sealed` modifier when you override `ToString` in a record type. Sealing the `ToString` method prevents the compiler from synthesizing a `ToString` method for any derived record types. A `sealed ToString` ensures all derived record types use the `ToString` method defined in a common base record type. You can learn more about this feature in the article on records.

Assignment and declaration in same deconstruction

This change removes a restriction from earlier versions of C#. Previously, a deconstruction could assign all values to existing variables, or initialize newly declared variables:

C#

Copy

```
// Initialization:
(int x, int y) = point;

// assignment:
int x1 = 0;
int y1 = 0;
(x1, y1) = point;
```

C# 10 removes this restriction:

C#

Copy

```
int x = 0;
(x, int y) = point;
```

Improved definite assignment

Prior to C# 10, there were many scenarios where definite assignment and null-state analysis produced warnings that were false positives. These generally involved comparisons to boolean constants, accessing a variable only in the `true` or `false` statements in an `if` statement, and null coalescing expressions. These examples generated warnings in previous versions of C#, but don't in C# 10:

C#

Copy

```
string representation = "N/A";
if ((c != null) && c.GetDependentValue(out object obj)) == true)
{
   representation = obj.ToString(); // undesired error
}

// Or, using ?.
if (c?.GetDependentValue(out object obj) == true)
{
   representation = obj.ToString(); // undesired error
}

// Or, using ??
if (c?.GetDependentValue(out object obj) ?? false)
{
   representation = obj.ToString(); // undesired error
}
```

The main impact of this improvement is that the warnings for definite assignment and null-state analysis are more accurate.

Allow AsyncMethodBuilder attribute on methods

In C# 10 and later, you can specify a different async method builder for a single method, in addition to specifying the method builder type for all methods that return a given task-like type. A custom async method builder enables advanced performance tuning scenarios where a given method may benefit from a custom builder.

To learn more, see the section on AsyncMethodBuilder in the article on attributes read by the compiler.

CallerArgumentExpression attribute diagnostics

You can use the System.Runtime.CompilerServices.CallerArgumentExpressionAttributeto specify a parameter that the compiler replaces with the text representation of another argument. This feature enables libraries to create more specific diagnostics. The following code tests a condition. If the condition is false, the exception message contains the text representation of the argument passed to `condition`:

C# Copy

```
public static void Validate(bool condition, [CallerArgumentExpression("condition")]
string? message=null)
{
    if (!condition)
    {
        throw new InvalidOperationException($"Argument failed validation:
<{message}>");
    }
}
```

You can learn more about this feature in the article on Caller information attributes in the language reference section.

Enhanced #line pragma

C# 10 supports a new format for the `#line` pragma. You likely won't use the new format, but you'll see its effects. The enhancements enable more fine-grained output in domain-specific languages (DSLs) like Razor. The Razor engine uses these enhancements to improve the debugging experience. You'll find debuggers can highlight your Razor source more accurately. To learn more about the new syntax, see the article on Preprocessor directives in the language reference. You can also read the feature specification for Razor based examples.

Generic attributes

Important

Generic attributes is a preview feature. You must **set <LangVersion> to Preview** to enable this feature. This feature may change before its final release.

You can declare a generic class whose base class is System.Attribute. This provides a more convenient syntax for attributes that require a System.Type parameter. Previously, you'd need to create an attribute that takes a `Type` as its constructor parameter:

C# Copy

```
public class TypeAttribute : Attribute
{
   public TypeAttribute(Type t) => ParamType = t;

   public Type ParamType { get; }
}
```

And to apply the attribute, you use the typeof operator:

C# Copy

```
[TypeAttribute(typeof(string))]
public string Method() => default;
```

Using this new feature, you can create a generic attribute instead:

C# Copy

```
public class GenericAttribute<T> : Attribute { }
```

Then, specify the type parameter to use the attribute:

C# Copy

```
[GenericAttribute<string>()]
public string Method() => default;
```

You can apply a fully closed constructed generic attribute. In other words, all type parameters must be specified. For example, the following is not allowed:

C# Copy

```
public class GenericType<T>
{
   [GenericAttribute<T>()] // Not allowed! generic attributes must be fully closed
types.
   public string Method() => default;
}
```

The type arguments must satisfy the same restrictions as the typeof operator. Types that require metadata annotations aren't allowed. Examples include the following:

- `dynamic`
- `nint, nuint`

- `string?` (or any nullable reference type)
- `(int X, int Y)` (or any other tuple types using C# tuple syntax).

These types aren't directly represented in metadata. They types include annotations that describe the type. In all cases, you can use the underlying type instead:

- `object` for `dynamic`.
- IntPtr instead of `nint` or `unint`.
- `string` instead of `string?`.
- `ValueTuple<int, int>` instead of `(int X, int Y)`.