What's new in C# 12

Article • 03/19/2024

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 operator `..`.

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 operator*, `..` in a collection expression replaces its argument with the elements from that collection. The argument must be a collection type. The following examples show how the spread operator 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 operand of a spread operator is an expression that can be enumerated. The spread operator evaluates each element of the enumerations expression.

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.

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