Breaking changes in Roslyn after .NET 9.0.100 through .NET 10.0.100

10/10/2025

This document lists known breaking changes in Roslyn after .NET 9 general release (.NET SDK version 9.0.100) through .NET 10 general release (.NET SDK version 10.0.100).

`scoped` in a lambda parameter list is now always a modifier.

Introduced in Visual Studio 2022 version 17.13

C# 14 introduces the ability to write a lambda with parameter modifiers, without having to specify a parameter type: Simple lambda parameters with modifiers

As part of this work, a breaking change was accepted where `scoped` will always be treated as a modifier in a lambda parameter, even where it might have been accepted as a type name in the past. For example:

```c#
var v = (scoped scoped s) => { ... };

ref struct @scoped { }
```

In C# 14 this will be an error as both `scoped` tokens are treated as modifiers. The workaround is to use `@` in the type name position like so:

```c#
var v = (scoped @scoped s) => { ... };

ref struct @scoped { }
```

`Span<T>` and `ReadOnlySpan<T>` overloads are applicable in more scenarios in C# 14 and newer

Introduced in Visual Studio 2022 version 17.13

C# 14 introduces new built-in span conversions and type inference rules . This means that different overloads might be chosen compared to C# 13, and sometimes an ambiguity compile-time error might be raised because a new overload is applicable but there is no single best overload.

The following example shows some ambiguities and possible workarounds. Note that another workaround is for API authors to use the `OverloadResolutionPriorityAttribute`.

C#

```csharp
var x = new long[] { 1 };
Assert.Equal([2], x); // previously Assert.Equal<T>(T[], T[]), now ambiguous with Assert.Equal<T>(ReadOnlySpan<T>, Span<T>)
Assert.Equal([2], x.AsSpan()); // workaround

var y = new int[] { 1, 2 };
var s = new ArraySegment<int>(y, 1, 1);
Assert.Equal(y, s); // previously Assert.Equal<T>(T, T), now ambiguous with Assert.Equal<T>(Span<T>, Span<T>)
Assert.Equal(y.AsSpan(), s); // workaround
```

A `Span<T>` overload might be chosen in C# 14 where an overload taking an interface implemented by `T[]` (such as `IEnumerable<T>`) was chosen in C# 13, and that can lead to an `ArrayTypeMismatchException` at runtime if used with a covariant array:

C#

```csharp
string[] s = new[] { "a" };
object[] o = s; // array variance

C.R(o); // wrote 1 previously, now crashes in Span<T> constructor with ArrayTypeMismatchException
C.R(o.AsEnumerable()); // workaround

static class C
{
    public static void R<T>(IEnumerable<T> e) => Console.Write(1);
    public static void R<T>(Span<T> s) => Console.Write(2);
    // another workaround:
    public static void R<T>(ReadOnlySpan<T> s) => Console.Write(3);
}
```

For that reason, `ReadOnlySpan<T>` is generally preferred over `Span<T>` by overload resolution in C# 14. In some cases, that might lead to compilation breaks, for example when there are overloads for both `Span<T>` and `ReadOnlySpan<T>`, both taking and returning the same span type:

cs

```cs
double[] x = new double[0];
Span<ulong> y = MemoryMarshal.Cast<double, ulong>(x); // previously worked, now compilation error
Span<ulong> z = MemoryMarshal.Cast<double, ulong>(x.AsSpan()); // work-around

static class MemoryMarshal
{
    public static ReadOnlySpan<TTo> Cast<TFrom, TTo>(ReadOnlySpan<TFrom> span) => default;
    public static Span<TTo> Cast<TFrom, TTo>(Span<TFrom> span) => default;
}
```

`Enumerable.Reverse`

When using C# 14 or newer and targeting a .NET older than `net10.0` or .NET Framework with `System.Memory` reference, there is a breaking change with `Enumerable.Reverse` and arrays.

> ⊗ **Caution**
>
> This only impacts customers using C# 14 and targeting .NET earlier than `net10.0`, which is an unsupported configuration.

C#

```csharp
int[] x = new[] { 1, 2, 3 };
var y = x.Reverse(); // previously Enumerable.Reverse, now MemoryExtensions.Reverse
```

On `net10.0`, there is `Enumerable.Reverse(this T[])` which takes precedence and hence the break is avoided. Otherwise, `MemoryExtensions.Reverse(this Span<T>)` is resolved which has different semantics than `Enumerable.Reverse(this IEnumerable<T>)` (which used to be resolved in C# 13 and lower). Specifically, the `Span` extension does the reversal in place and returns `void`. As a workaround, one can define their own `Enumerable.Reverse(this T[])` or use `Enumerable.Reverse` explicitly:

C#

```csharp
int[] x = new[] { 1, 2, 3 };
var y = Enumerable.Reverse(x); // instead of 'x.Reverse();'
```

Diagnostics now reported for pattern-based disposal method in `foreach`

Introduced in Visual Studio 2022 version 17.13

For instance, an obsolete `DisposeAsync` method is now reported in `await foreach`.

C#

```csharp
await foreach (var i in new C()) { } // 'C.AsyncEnumerator.DisposeAsync()' is obsolete

class C
{
    public AsyncEnumerator GetAsyncEnumerator(System.Threading.CancellationToken token = default)
    {
        throw null;
    }

    public sealed class AsyncEnumerator : System.IAsyncDisposable
    {
        public int Current { get => throw null; }
        public Task<bool> MoveNextAsync() => throw null;

        [System.Obsolete]
        public ValueTask DisposeAsync() => throw null;
    }
}
```

Set state of enumerator object to "after" during disposal

Introduced in Visual Studio 2022 version 17.13

The state machine for enumerators incorrectly allowed resuming execution after the enumerator was disposed.
Now, `MoveNext()` on a disposed enumerator properly returns `false` without executing any more user code.

C#

```csharp
var enumerator = C.GetEnumerator();

Console.Write(enumerator.MoveNext()); // prints True
Console.Write(enumerator.Current); // prints 1
```

```
enumerator.Dispose();

Console.Write(enumerator.MoveNext()); // now prints False

class C
{
    public static IEnumerator<int> GetEnumerator()
    {
        yield return 1;
        Console.Write("not executed after disposal")
        yield return 2;
    }
}
```

Warn for redundant pattern in simple `or` patterns

Introduced in Visual Studio 2022 version 17.13

In a disjunctive `or` pattern such as `is not null or 42` or `is not int or string` the second pattern is redundant and likely results from misunderstanding the precedence order of `not` and `or` pattern combinators.
The compiler provides a warning in common cases of this mistake:

C#

```
_ = o is not null or 42; // warning: pattern "42" is redundant
_ = o is not int or string; // warning: pattern "string" is redundant
```

It is likely that the user meant `is not (null or 42)` or `is not (int or string)` instead.

`UnscopedRefAttribute` cannot be used with old ref safety rules

Introduced in Visual Studio 2022 version 17.13

The `UnscopedRefAttribute` unintentionally affected code compiled by new Roslyn compiler versions even when the code was compiled in the context of the earlier ref safety rules (i.e., targeting C# 10 or earlier with net6.0 or earlier). However, the attribute should not have an effect in that context, and that is now fixed.

Code that previously did not report any errors in C# 10 or earlier with net6.0 or earlier can now fail to compile:

C#

```csharp
using System.Diagnostics.CodeAnalysis;
struct S
{
    public int F;

    // previously allowed in C# 10 with net6.0
    // now fails with the same error as if the [UnscopedRef] wasn't there:
    // error CS8170: Struct members cannot return 'this' or other instance members by reference
    [UnscopedRef] public ref int Ref() => ref F;
}
```

To prevent misunderstanding (thinking the attribute has an effect but it actually does not because your code is compiled with the earlier ref safety rules), a warning is reported when the attribute is used in C# 10 or earlier with net6.0 or earlier:

C#

```csharp
using System.Diagnostics.CodeAnalysis;
struct S
{
    // both are errors in C# 10 with net6.0:
    // warning CS9269: UnscopedRefAttribute is only valid in C# 11 or later or when targeting net7.0 or later.
    [UnscopedRef] public ref int Ref() => throw null!;
    public static void M([UnscopedRef] ref int x) { }
}
```

`Microsoft.CodeAnalysis.EmbeddedAttribute` is validated on declaration

Introduced in Visual Studio 2022 version 17.13

The compiler now validates the shape of `Microsoft.CodeAnalysis.EmbeddedAttribute` when declared in source. Previously, the compiler would allow user-defined declarations of this attribute, but only when it didn't need to generate one itself. We now validate that:

1. It must be internal
2. It must be a class
3. It must be sealed
4. It must be non-static
5. It must have an internal or public parameterless constructor
6. It must inherit from System.Attribute.

7. It must be allowed on any type declaration (class, struct, interface, enum, or delegate)

cs

```cs
namespace Microsoft.CodeAnalysis;

// Previously, sometimes allowed. Now, CS9271
public class EmbeddedAttribute : Attribute {}
```

Expression `field` in a property accessor refers to synthesized backing field

Introduced in Visual Studio 2022 version 17.12

The expression `field`, when used within a property accessor, refers to a synthesized backing field for the property.

The warning CS9258 is reported when the identifier would have bound to a different symbol with language version 13 or earlier.

To avoid generating a synthesized backing field, and to refer to the existing member, use 'this.field' or '@field' instead. Alternatively, rename the existing member and the reference to that member to avoid a conflict with `field`.

C#

```csharp
class MyClass
{
    private int field = 0;

    public object Property
    {
        get
        {
            // warning CS9258: The 'field' keyword binds to a synthesized backing field for the property.
            // To avoid generating a synthesized backing field, and to refer to the existing member,
            // use 'this.field' or '@field' instead.
            return field;
        }
    }
}
```

Variable named `field` disallowed in a property accessor

Introduced in Visual Studio 2022 version 17.14

The expression `field`, when used within a property accessor, refers to a synthesized backing field for the property.

The error CS9272 is reported when a local, or a parameter of a nested function, with the name `field` is declared in a property accessor.

To avoid the error, rename the variable, or use `@field` in the declaration.

C#

```csharp
class MyClass
{
    public object Property
    {
        get
        {
            // error CS9272: 'field' is a keyword within a property accessor.
            // Rename the variable or use the identifier '@field' instead.
            int field = 0;
            return @field;
        }
    }
}
```

`record` and `record struct` types cannot define pointer type members, even when providing their own Equals implementations

Introduced in Visual Studio 2022 version 17.14

The specification for `record class` and `record struct` types indicated that any pointer types are disallowed as instance fields. However, this was not enforced correctly when the `record class` or `record struct` type defined its own `Equals` implementation.

The compiler now correctly forbids this.

CS

```
unsafe record struct R(
    int* P // Previously fine, now CS8908
)
{
    public bool Equals(R other) => true;
}
```

Emitting metadata-only executables requires an entrypoint

Introduced in Visual Studio 2022 version 17.14

Previously, the entrypoint was unintentionally unset when emitting executables in metadata-only mode (also known as ref assemblies). That is now corrected but it also means that a missing entrypoint is a compilation error:

```cs
// previously successful, now fails:
CSharpCompilation.Create("test").Emit(new MemoryStream(),
    options: EmitOptions.Default.WithEmitMetadataOnly(true))

CSharpCompilation.Create("test",
    // workaround - mark as DLL instead of EXE (the default):
    options: new
CSharpCompilationOptions(OutputKind.DynamicallyLinkedLibrary))
    .Emit(new MemoryStream(),
        options: EmitOptions.Default.WithEmitMetadataOnly(true))
```

Similarly this can be observed when using the command-line argument `/refonly` or the `ProduceOnlyReferenceAssembly` MSBuild property.

`partial` cannot be a return type of methods

Introduced in Visual Studio 2022 version 17.14

The partial events and constructors language feature allows the `partial` modifier in more places and so it cannot be a return type unless escaped:

```cs
class C
{
    partial F() => new partial(); // previously worked
    @partial F() => new partial(); // workaround
}

class partial { }
```

`extension` treated as a contextual keyword

Introduced in Visual Studio 2022 version 17.14. Starting with C# 14, the `extension` keyword serves a special purpose in denoting extension containers. This changes how the compiler interprets certain code constructs.

If you need to use "extension" as an identifier rather than a keyword, escape it with the `@` prefix: `@extension`. This tells the compiler to treat it as a regular identifier instead of a keyword.

The compiler will parse this as an extension container rather than a constructor.

C#

```
class @extension
{
    extension(object o) { } // parsed as an extension container
}
```

The compiler will fail to parse this as a method with return type `extension`.

C#

```
class @extension
{
    extension M() { } // will not compile
}
```

Introduced in Visual Studio 2026 version 18.0. The "extension" identifier may not be used as a type name, so the following will not compile:

C#

```
using extension = ...; // alias may not be named "extension"
class extension { } // type may not be named "extension"
class C<extension> { } // type parameter may not be named "extension"
```

Partial properties and events are now implicitly virtual and public

Introduced in Visual Studio 2026 version 18.0 preview 1

We have fixed an inconsistency where partial interface properties and events would not be implicitly `virtual` and `public` unlike their non-partial equivalents. This inconsistency is however preserved for partial interface methods to avoid a larger breaking change. Note that Visual Basic and other languages not supporting default interface members will start requiring to implement implicitly virtual `partial` interface members.

To keep the previous behavior, explicitly mark `partial` interface members as `private` (if they don't have any accessibility modifiers) and `sealed` (if they don't have the `private` modifier which implies `sealed`, and they don't already have modifier `virtual` or `sealed`).

```cs
System.Console.Write(((I)new C()).P); // wrote 1 previously, writes 2 now

partial interface I
{
    public partial int P { get; }
    public partial int P => 1; // implicitly virtual now
}

class C : I
{
    public int P => 2; // implements I.P
}
```

```cs
System.Console.Write(((I)new C()).P); // inaccessible previously, writes 1
now

partial interface I
{
    partial int P { get; } // implicitly public now
    partial int P => 1;
}

class C : I;
```

Missing `ParamCollectionAttribute` is reported in more cases

Introduced in Visual Studio 2026 version 18.0

If you are compiling a `.netmodule` (note that this doesn't apply to normal DLL/EXE compilations), and have a lambda or a local function with a `params` collection parameter, and the `ParamCollectionAttribute` is not found, a compilation error is now reported (because the attribute now must be emitted on the synthesized method but the attribute type itself is not synthesized by the compiler into a `.netmodule`). You can work around that by defining the attribute yourself.

cs

```
using System;
using System.Collections.Generic;
class C
{
    void M()
    {
        Func<IList<int>, int> lam = (params IList<int> xs) => xs.Count; // error if ParamCollectionAttribute does not exist
        lam([1, 2, 3]);

        int func(params IList<int> xs) => xs.Count; // error if ParamCollectionAttribute does not exist
        func(4, 5, 6);
    }
}
```