Tell us about your PDF experience.

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

What's new in C# 11

REFERENCE

Breaking changes in the C# compiler

Version compatibility

Stay in touch

REFERENCE

.NET developer community

YouTube

Twitter

C# language versioning

Article • 10/08/2024

The latest C# compiler determines a default language version based on your project's target framework or frameworks. Visual Studio doesn't provide a UI to change the value, but you can change it by editing the *csproj* file. The choice of default ensures that you use the latest language version compatible with your target framework. You benefit from access to the latest language features compatible with your project's target. This default choice also ensures you don't use a language that requires types or runtime behavior not available in your target framework. Choosing a language version newer than the default can cause hard to diagnose compile-time and runtime errors.

C# 13 is supported only on .NET 9 and newer versions. C# 12 is supported only on .NET 8 and newer versions. C# 11 is supported only on .NET 7 and newer versions. Using a C# language version newer than the version associated with your target TFM is unsupported.

Check the Visual Studio platform compatibility page for details on which .NET versions are supported by versions of Visual Studio. Check the Mono page for C# for Mono compatibility with C# versions.

Defaults

The compiler determines a default based on these rules:

Expand table

Target	Version	C# language version default
.NET	10.x	C# 14
.NET	9.x	C# 13
.NET	8.x	C# 12
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3

Target	Version	C# language version default
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

If your project targets a `preview` framework that has a corresponding preview language version, the language version used is the preview language version. You use the latest features with that preview in any environment, without affecting projects that target a released .NET Core version.

C# language version reference

The following table shows all current C# language versions. Older compilers might not understand every value. If you install the latest .NET SDK, you have access to everything listed.

Expand table

Value	Meaning
`preview`	The compiler accepts all valid language syntax from the latest preview version.
`latest`	The compiler accepts syntax from the latest released version of the compiler (including minor version).
`latestMajor` or `default`	The compiler accepts syntax from the latest released major version of the compiler.
`14.0`	The compiler accepts only syntax that is included in C# 14 or lower.
`13.0`	The compiler accepts only syntax that is included in C# 13 or lower.
`12.0`	The compiler accepts only syntax that is included in C# 12 or lower.
`11.0`	The compiler accepts only syntax that is included in C# 11 or lower.
`10.0`	The compiler accepts only syntax that is included in C# 10 or lower.
`9.0`	The compiler accepts only syntax that is included in C# 9 or lower.
`8.0`	The compiler accepts only syntax that is included in C# 8.0 or lower.
`7.3`	The compiler accepts only syntax that is included in C# 7.3 or lower.

Value	Meaning
`7.2`	The compiler accepts only syntax that is included in C# 7.2 or lower.
`7.1`	The compiler accepts only syntax that is included in C# 7.1 or lower.
`7`	The compiler accepts only syntax that is included in C# 7.0 or lower.
`6`	The compiler accepts only syntax that is included in C# 6.0 or lower.
`5`	The compiler accepts only syntax that is included in C# 5.0 or lower.
`4`	The compiler accepts only syntax that is included in C# 4.0 or lower.
`3`	The compiler accepts only syntax that is included in C# 3.0 or lower.
`ISO-2` or `2`	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
`ISO-1` or `1`	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

> **Note**
>
> Specifying **LangVersion** with the `default` value is different from omitting the **LangVersion** option. Specifying `default` uses the latest version of the language that the compiler supports, without taking into account the target framework. For example, building a project that targets .NET 6 from the current version of Visual Studio 2022 uses C# 10 if **LangVersion** isn't specified, but uses C# 12 if **LangVersion** is set to `default`.

Configure C# language version

Article • 01/31/2025

The information in this article applies to .NET 5 and above. For UWP projects, see this information in the article on Choosing a UWP version.

In Visual Studio, the option to change the language version through the UI is disabled because the default version is aligned with the project's target framework (`TFM`). This default configuration ensures compatibility between language features and runtime support. To change the language version in Visual Studio, change the project's target framework.

For example, changing the target `TFM` (for example, from .NET 6 to .NET 9) updates the language version accordingly, from C# 10 to C# 13. This approach prevents issues with runtime compatibility and minimizes unexpected build errors due to unsupported language features.

If you need a specific language version that differs from the one automatically selected, refer to the methods in this article to override the default settings directly in the project file.

> **⚠ Warning**
>
> Setting the `LangVersion` element to `latest` is discouraged. The `latest` setting means the installed compiler uses its latest version. The value of `latest` can change from machine to machine, making builds unreliable. In addition, it enables language features that might require runtime or library features not included in the current SDK.

If you must specify your C# version explicitly, you can do so in several ways:

- Manually edit the project file.
- Set the language version for multiple projects in a subdirectory.
- Configure the LangVersion compiler option.

> **Tip**
>
> You can see the language version in Visual Studio in the project properties page. Under the *Build* tab, the *Advanced* pane displays the version selected.

To know what language version you're currently using, put `#error version` (case sensitive) in your code. This pragma makes the compiler report a compiler error, CS8304, with a message containing the compiler version and the current selected language version. For more information about this pragma, see #error (C# Reference).

Edit the project file

You can set the language version in your project file. For example, if you explicitly want access to preview features, add an element like the following example:

XML

```xml
<PropertyGroup>
   <LangVersion>preview</LangVersion>
</PropertyGroup>
```

The value `preview` uses the latest available preview C# language version your compiler supports.

Configure multiple projects

To configure multiple C# projects, you can create a *Directory.Build.props* file, typically in your solution directory, that contains the `<LangVersion>` element. Add the following setting to the *Directory.Build.props* file:

XML

```xml
<Project>
 <PropertyGroup>
   <LangVersion>preview</LangVersion>
 </PropertyGroup>
</Project>
```

Builds in all subdirectories of the directory containing that file now use the preview C# version. For more information, see Customize your build.

> **Note**
>
> The versions for C# and VB are different. Don't use the *Directory.Build.Props* file for a folder where subdirectories contain projects for both languages. The versions

don't match.

C# language version reference

> **Important**
>
> Using a C# language version newer than the version associated with your target TFM is unsupported.

The following table shows all current C# language versions. Older compilers might not understand every value. If you install the latest .NET SDK, you have access to everything listed.

Expand table

Value	Meaning
`preview`	The compiler accepts all valid language syntax from the latest preview version.
`latest`	The compiler accepts syntax from the latest released version of the compiler (including minor version).
`latestMajor` or `default`	The compiler accepts syntax from the latest released major version of the compiler.
`14.0`	The compiler accepts only syntax that is included in C# 14 or lower.
`13.0`	The compiler accepts only syntax that is included in C# 13 or lower.
`12.0`	The compiler accepts only syntax that is included in C# 12 or lower.
`11.0`	The compiler accepts only syntax that is included in C# 11 or lower.
`10.0`	The compiler accepts only syntax that is included in C# 10 or lower.
`9.0`	The compiler accepts only syntax that is included in C# 9 or lower.
`8.0`	The compiler accepts only syntax that is included in C# 8.0 or lower.
`7.3`	The compiler accepts only syntax that is included in C# 7.3 or lower.
`7.2`	The compiler accepts only syntax that is included in C# 7.2 or lower.
`7.1`	The compiler accepts only syntax that is included in C# 7.1 or lower.
`7`	The compiler accepts only syntax that is included in C# 7.0 or lower.

Value	Meaning
`6`	The compiler accepts only syntax that is included in C# 6.0 or lower.
`5`	The compiler accepts only syntax that is included in C# 5.0 or lower.
`4`	The compiler accepts only syntax that is included in C# 4.0 or lower.
`3`	The compiler accepts only syntax that is included in C# 3.0 or lower.
`ISO-2` or `2`	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
`ISO-1` or `1`	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

Built-in types (C# reference)

08/05/2025

The following table lists the C# built-in value types:

Expand table

C# type keyword	.NET type
bool	System.Boolean
byte	System.Byte
sbyte	System.SByte
char	System.Char
decimal	System.Decimal
double	System.Double
float	System.Single
int	System.Int32
uint	System.UInt32
nint	System.IntPtr
nuint	System.UIntPtr
long	System.Int64
ulong	System.UInt64
short	System.Int16
ushort	System.UInt16

The following table lists the C# built-in reference types:

Expand table

C# type keyword	.NET type
object	System.Object
string	System.String
delegate	System.Delegate

C# type keyword	.NET type
dynamic	System.Object

In the preceding tables, most C# type keywords from the left column are aliases for the corresponding .NET type. They're interchangeable. For example, the following declarations declare variables of the same type:

C#

```
int a = 123;
System.Int32 b = 123;
```

The `dynamic` type is similar to `object`. The main differences are:

- Operations on a `dynamic` expression are bound at runtime, not at compile time.
- You can't use `new dynamic()`.
- You can't derive a type from the `dynamic` type.

The `delegate` keyword is a built-in reference type keyword that declares a type derived from System.Delegate. Unlike the other built-in type keywords, `delegate` isn't an alias for a specific .NET type. Instead, it declares custom types that derive from the abstract `System.Delegate` type. Similarly, `dynamic` represents runtime binding behavior rather than being a direct alias for a specific .NET type.

The void keyword represents the absence of a type. You use it as the return type of a method that doesn't return a value.

The C# language includes specialized rules for the System.Span<T> and System.ReadOnlySpan<T> types. These types aren't classified as built-in types, because there aren't C# keywords that correspond to these types. The C# language defines implicit conversions from array types and the string type to `Span<T>` and `ReadOnlySpan<T>`. These conversions integrate `Span` types into more natural programming scenarios. The following conversions are defined as *implicit span conversions*:

- From any single-dimensional array with element type `E` to `System.Span<E>`
- From any single-dimensional array with element type `E` to `System.ReadOnlySpan<U>`, when `E` has covariance conversion or an identity conversion to `U`
- From `System.Span<E>` to `System.ReadOnlySpan<U>`, when `E` has a covariance conversion or an identity conversion to `U`
- From `System.ReadOnlySpan<E>` to `System.ReadOnlySpan<U>`, when `E` has a covariance conversion or an identity conversion to `U`

- From `string` to `System.ReadOnlySpan<char>`

The compiler never ignores any user defined conversion where an applicable *implicit span conversion* exists. Implicit span conversions can be applied to the first argument of extension methods, the parameter with the `this` modifier. Implicit span conversions aren't considered for method group conversions.

See also

- Use language keywords instead of framework type names (style rule IDE0049)
- Default values of C# types

Value types (C# reference)

Article • 10/25/2024

Value types and reference types are the two main categories of C# types. A variable of a value type contains an instance of the type. This differs from a variable of a reference type, which contains a reference to an instance of the type. By default, on assignment, passing an argument to a method, and returning a method result, variable values are copied. In the case of value-type variables, the corresponding type instances are copied. The following example demonstrates that behavior:

C#

```
using System;

public struct MutablePoint
{
    public int X;
    public int Y;

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

    public override string ToString() => $"({X}, {Y})";
}

public class Program
{
    public static void Main()
    {
        var p1 = new MutablePoint(1, 2);
        var p2 = p1;
        p2.Y = 200;
        Console.WriteLine($"{nameof(p1)} after {nameof(p2)} is modified:
{p1}");
        Console.WriteLine($"{nameof(p2)}: {p2}");

        MutateAndDisplay(p2);
        Console.WriteLine($"{nameof(p2)} after passing to a method: {p2}");
    }

    private static void MutateAndDisplay(MutablePoint p)
    {
        p.X = 100;
        Console.WriteLine($"Point mutated in a method: {p}");
    }
}
// Expected output:
// p1 after p2 is modified: (1, 2)
// p2: (1, 200)
```

```
// Point mutated in a method: (100, 200)
// p2 after passing to a method: (1, 200)
```

As the preceding example shows, operations on a value-type variable affect only that instance of the value type, stored in the variable.

If a value type contains a data member of a reference type, only the reference to the instance of the reference type is copied when a value-type instance is copied. Both the copy and original value-type instance have access to the same reference-type instance. The following example demonstrates that behavior:

C#

```
using System;
using System.Collections.Generic;

public struct TaggedInteger
{
    public int Number;
    private List<string> tags;

    public TaggedInteger(int n)
    {
        Number = n;
        tags = new List<string>();
    }

    public void AddTag(string tag) => tags.Add(tag);

    public override string ToString() => $"{Number} [{string.Join(", ",
tags)}]";
}

public class Program
{
    public static void Main()
    {
        var n1 = new TaggedInteger(0);
        n1.AddTag("A");
        Console.WriteLine(n1);  // output: 0 [A]

        var n2 = n1;
        n2.Number = 7;
        n2.AddTag("B");

        Console.WriteLine(n1);  // output: 0 [A, B]
        Console.WriteLine(n2);  // output: 7 [A, B]
    }
}
```

> **Note**
>
> To make your code less error-prone and more robust, define and use immutable value types. This article uses mutable value types only for demonstration purposes.

Kinds of value types and type constraints

A value type can be one of the two following kinds:

- a structure type, which encapsulates data and related functionality
- an enumeration type, which is defined by a set of named constants and represents a choice or a combination of choices

A nullable value type `T?` represents all values of its underlying value type `T` and an additional null value. You cannot assign `null` to a variable of a value type, unless it's a nullable value type.

You can use the struct constraint to specify that a type parameter is a non-nullable value type. Both structure and enumeration types satisfy the `struct` constraint. You can use `System.Enum` in a base class constraint (that is known as the enum constraint) to specify that a type parameter is an enumeration type.

Built-in value types

C# provides the following built-in value types, also known as *simple types*:

- Integral numeric types
- Floating-point numeric types
- bool that represents a Boolean value
- char that represents a Unicode UTF-16 character

All simple types are structure types and differ from other structure types in that they permit certain additional operations:

- You can use literals to provide a value of a simple type.
  For example, `'A'` is a literal of the type `char`, `2001` is a literal of the type `int` and `12.34m` is a literal of the type `decimal`.

- You can declare constants of the simple types with the const keyword.
  For example, you can define `const decimal = 12.34m`.
  It's not possible to have constants of other structure types.

- Constant expressions, whose operands are all constants of the simple types, are evaluated at compile time.

A value tuple is a value type, but not a simple type.

C# language specification

For more information, see the following sections of the C# language specification:

- Value types
- Simple types
- Variables

See also

- System.ValueType
- Reference types

Integral numeric types (C# reference)

Article • 09/29/2022

The *integral numeric types* represent integer numbers. All integral numeric types are value types. They're also simple types and can be initialized with literals. All integral numeric types support arithmetic, bitwise logical, comparison, and equality operators.

Characteristics of the integral types

C# supports the following predefined integral types:

Expand table

C# type/keyword	Range	Size	.NET type
`sbyte`	-128 to 127	Signed 8-bit integer	System.SByte
`byte`	0 to 255	Unsigned 8-bit integer	System.Byte
`short`	-32,768 to 32,767	Signed 16-bit integer	System.Int16
`ushort`	0 to 65,535	Unsigned 16-bit integer	System.UInt16
`int`	-2,147,483,648 to 2,147,483,647	Signed 32-bit integer	System.Int32
`uint`	0 to 4,294,967,295	Unsigned 32-bit integer	System.UInt32
`long`	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	Signed 64-bit integer	System.Int64
`ulong`	0 to 18,446,744,073,709,551,615	Unsigned 64-bit integer	System.UInt64
`nint`	Depends on platform (computed at runtime)	Signed 32-bit or 64-bit integer	System.IntPtr
`nuint`	Depends on platform (computed at runtime)	Unsigned 32-bit or 64-bit integer	System.UIntPtr

In all of the table rows except the last two, each C# type keyword from the leftmost column is an alias for the corresponding .NET type. The keyword and .NET type name are interchangeable. For example, the following declarations declare variables of the same type:

```
C#
```

```
int a = 123;
System.Int32 b = 123;
```

The `nint` and `nuint` types in the last two rows of the table are native-sized integers. You can use the `nint` and `nuint` contextual keywords to define *native-sized integers*. These are 32-bit integers when running in a 32-bit process, or 64-bit integers when running in a 64-bit process. They can be used for interop scenarios, low-level libraries, and to optimize performance in scenarios where integer math is used extensively.

The native-sized integer types are represented internally as the .NET types System.IntPtr and System.UIntPtr. Starting in C# 11, the `nint` and `nuint` types are aliases for the underlying types.

The default value of each integral type is zero, `0`.

Each of the integral types has `MinValue` and `MaxValue` properties that provide the minimum and maximum value of that type. These properties are compile-time constants except for the case of the native-sized types (`nint` and `nuint`). The `MinValue` and `MaxValue` properties are calculated at runtime for native-sized types. The sizes of those types depend on the process settings.

Use the System.Numerics.BigInteger structure to represent a signed integer with no upper or lower bounds.

Integer literals

Integer literals can be

- *decimal*: without any prefix
- *hexadecimal*: with the `0x` or `0X` prefix
- *binary*: with the `0b` or `0B` prefix

The following code demonstrates an example of each:

```
C#
```

```
var decimalLiteral = 42;
var hexLiteral = 0x2A;
var binaryLiteral = 0b_0010_1010;
```

The preceding example also shows the use of `_` as a *digit separator*. You can use the digit separator with all kinds of numeric literals.

The type of an integer literal is determined by its suffix as follows:

- If the literal has no suffix, its type is the first of the following types in which its value can be represented: `int`, `uint`, `long`, `ulong`.

  > **Note**
  >
  > Literals are interpreted as positive values. For example, the literal `0xFF_FF_FF_FF` represents the number `4294967295` of the `uint` type, though it has the same bit representation as the number `-1` of the `int` type. If you need a value of a certain type, cast a literal to that type. Use the `unchecked` operator, if a literal value cannot be represented in the target type. For example, `unchecked((int)0xFF_FF_FF_FF)` produces `-1`.

- If the literal is suffixed by `U` or `u`, its type is the first of the following types in which its value can be represented: `uint`, `ulong`.

- If the literal is suffixed by `L` or `l`, its type is the first of the following types in which its value can be represented: `long`, `ulong`.

  > **Note**
  >
  > You can use the lowercase letter `l` as a suffix. However, this generates a compiler warning because the letter `l` can be confused with the digit `1`. Use `L` for clarity.

- If the literal is suffixed by `UL`, `Ul`, `uL`, `ul`, `LU`, `Lu`, `lU`, or `lu`, its type is `ulong`.

If the value represented by an integer literal exceeds UInt64.MaxValue, a compiler error CS1021 occurs.

If the determined type of an integer literal is `int` and the value represented by the literal is within the range of the destination type, the value can be implicitly converted to

`sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint` or `nuint`:

```
C#

byte a = 17;
byte b = 300;   // CS0031: Constant value '300' cannot be converted to a
'byte'
```

As the preceding example shows, if the literal's value isn't within the range of the destination type, a compiler error CS0031 occurs.

You can also use a cast to convert the value represented by an integer literal to the type other than the determined type of the literal:

```
C#

var signedByte = (sbyte)42;
var longVariable = (long)42;
```

Conversions

You can convert any integral numeric type to any other integral numeric type. If the destination type can store all values of the source type, the conversion is implicit. Otherwise, you need to use a cast expression to perform an explicit conversion. For more information, see Built-in numeric conversions.

Native sized integers

Native sized integer types have special behavior because the storage is determined by the natural integer size on the target machine.

- To get the size of a native-sized integer at run time, you can use `sizeof()`. However, the code must be compiled in an unsafe context. For example:

  ```
  C#

  Console.WriteLine($"size of nint = {sizeof(nint)}");
  Console.WriteLine($"size of nuint = {sizeof(nuint)}");

  // output when run in a 64-bit process
  //size of nint = 8
  //size of nuint = 8

  // output when run in a 32-bit process
  ```

```
//size of nint = 4
//size of nuint = 4
```

You can also get the equivalent value from the static IntPtr.Size and UIntPtr.Size properties.

- To get the minimum and maximum values of native-sized integers at run time, use `MinValue` and `MaxValue` as static properties with the `nint` and `nuint` keywords, as in the following example:

```
C#
```

```
Console.WriteLine($"nint.MinValue = {nint.MinValue}");
Console.WriteLine($"nint.MaxValue = {nint.MaxValue}");
Console.WriteLine($"nuint.MinValue = {nuint.MinValue}");
Console.WriteLine($"nuint.MaxValue = {nuint.MaxValue}");

// output when run in a 64-bit process
//nint.MinValue = -9223372036854775808
//nint.MaxValue = 9223372036854775807
//nuint.MinValue = 0
//nuint.MaxValue = 18446744073709551615

// output when run in a 32-bit process
//nint.MinValue = -2147483648
//nint.MaxValue = 2147483647
//nuint.MinValue = 0
//nuint.MaxValue = 4294967295
```

- You can use constant values in the following ranges:
  - For `nint`: Int32.MinValue to Int32.MaxValue.
  - For `nuint`: UInt32.MinValue to UInt32.MaxValue.

- The compiler provides implicit and explicit conversions to other numeric types. For more information, see Built-in numeric conversions.

- There's no direct syntax for native-sized integer literals. There's no suffix to indicate that a literal is a native-sized integer, such as `L` to indicate a `long`. You can use implicit or explicit casts of other integer values instead. For example:

```
C#
```

```
nint a = 42
nint a = (nint)42;
```

C# language specification

For more information, see the following sections of the C# language specification:

- Integral types
- Integer literals
- Native sized integral types
- C# 11 - Numeric IntPtr and UIntPtr

See also

- Value types
- Floating-point types
- Standard numeric format strings
- Numerics in .NET

Floating-point numeric types (C# reference)

Article • 09/29/2022

The *floating-point numeric types* represent real numbers. All floating-point numeric types are value types. They are also simple types and can be initialized with literals. All floating-point numeric types support arithmetic, comparison, and equality operators.

Characteristics of the floating-point types

C# supports the following predefined floating-point types:

Expand table

C# type/keyword	Approximate range	Precision	Size	.NET type
`float`	±1.5 x 10^{-45} to ±3.4 x 10^{38}	~6-9 digits	4 bytes	System.Single
`double`	±5.0 × 10^{-324} to ±1.7 × 10^{308}	~15-17 digits	8 bytes	System.Double
`decimal`	±1.0 x 10^{-28} to ±7.9228 x 10^{28}	28-29 digits	16 bytes	System.Decimal

In the preceding table, each C# type keyword from the leftmost column is an alias for the corresponding .NET type. They are interchangeable. For example, the following declarations declare variables of the same type:

```
C#

double a = 12.3;
System.Double b = 12.3;
```

The default value of each floating-point type is zero, `0`. Each of the floating-point types has the `MinValue` and `MaxValue` constants that provide the minimum and maximum finite value of that type. The `float` and `double` types also provide constants that represent not-a-number and infinity values. For example, the `double` type provides the following constants: Double.NaN, Double.NegativeInfinity, and Double.PositiveInfinity.

The `decimal` type is appropriate when the required degree of precision is determined by the number of digits to the right of the decimal point. Such numbers are commonly used in financial applications, for currency amounts (for example, $1.00), interest rates (for example, 2.625%), and so forth. Even numbers that are precise to only one decimal

digit are handled more accurately by the `decimal` type: 0.1, for example, can be exactly represented by a `decimal` instance, while there's no `double` or `float` instance that exactly represents 0.1. Because of this difference in numeric types, unexpected rounding errors can occur in arithmetic calculations when you use `double` or `float` for decimal data. You can use `double` instead of `decimal` when optimizing performance is more important than ensuring accuracy. However, any difference in performance would go unnoticed by all but the most calculation-intensive applications. Another possible reason to avoid `decimal` is to minimize storage requirements. For example, ML.NET uses `float` because the difference between 4 bytes and 16 bytes adds up for very large data sets. For more information, see System.Decimal.

You can mix integral types and the `float` and `double` types in an expression. In this case, integral types are implicitly converted to one of the floating-point types and, if necessary, the `float` type is implicitly converted to `double`. The expression is evaluated as follows:

- If there is `double` type in the expression, the expression evaluates to `double`, or to bool in relational and equality comparisons.
- If there is no `double` type in the expression, the expression evaluates to `float`, or to `bool` in relational and equality comparisons.

You can also mix integral types and the `decimal` type in an expression. In this case, integral types are implicitly converted to the `decimal` type and the expression evaluates to `decimal`, or to `bool` in relational and equality comparisons.

You cannot mix the `decimal` type with the `float` and `double` types in an expression. In this case, if you want to perform arithmetic, comparison, or equality operations, you must explicitly convert the operands either from or to the `decimal` type, as the following example shows:

C#

```
double a = 1.0;
decimal b = 2.1m;
Console.WriteLine(a + (double)b);
Console.WriteLine((decimal)a + b);
```

You can use either standard numeric format strings or custom numeric format strings to format a floating-point value.

Real literals

The type of a real literal is determined by its suffix as follows:

- The literal without suffix or with the `d` or `D` suffix is of type `double`
- The literal with the `f` or `F` suffix is of type `float`
- The literal with the `m` or `M` suffix is of type `decimal`

The following code demonstrates an example of each:

```
C#
```

```
double d = 3D;
d = 4d;
d = 3.934_001;

float f = 3_000.5F;
f = 5.4f;

decimal myMoney = 3_000.5m;
myMoney = 400.75M;
```

The preceding example also shows the use of `_` as a *digit separator*. You can use the digit separator with all kinds of numeric literals.

You can also use scientific notation, that is, specify an exponent part of a real literal, as the following example shows:

```
C#
```

```
double d = 0.42e2;
Console.WriteLine(d);  // output 42

float f = 134.45E-2f;
Console.WriteLine(f);  // output: 1.3445

decimal m = 1.5E6m;
Console.WriteLine(m);  // output: 1500000
```

Conversions

There is only one implicit conversion between floating-point numeric types: from `float` to `double`. However, you can convert any floating-point type to any other floating-point type with the explicit cast. For more information, see Built-in numeric conversions.

C# language specification

For more information, see the following sections of the C# language specification:

- Floating-point types
- The decimal type
- Real literals

See also

- Value types
- Integral types
- Standard numeric format strings
- Numerics in .NET
- System.Numerics.Complex

Built-in numeric conversions (C# reference)

Article • 01/31/2023

C# provides a set of integral and floating-point numeric types. There exists a conversion between any two numeric types, either implicit or explicit. You must use a cast expression to perform an explicit conversion.

Implicit numeric conversions

The following table shows the predefined implicit conversions between the built-in numeric types:

Expand table

From	To
sbyte	`short`, `int`, `long`, `float`, `double`, `decimal`, or `nint`
byte	`short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `decimal`, `nint`, or `nuint`
short	`int`, `long`, `float`, `double`, or `decimal`, or `nint`
ushort	`int`, `uint`, `long`, `ulong`, `float`, `double`, or `decimal`, `nint`, or `nuint`
int	`long`, `float`, `double`, or `decimal`, `nint`
uint	`long`, `ulong`, `float`, `double`, or `decimal`, or `nuint`
long	`float`, `double`, or `decimal`
ulong	`float`, `double`, or `decimal`
float	`double`
nint	`long`, `float`, `double`, or `decimal`
nuint	`ulong`, `float`, `double`, or `decimal`

> **Note**
>
> The implicit conversions from `int`, `uint`, `long`, `ulong`, `nint`, or `nuint` to `float` and from `long`, `ulong`, `nint`, or `nuint` to `double` may cause a loss of precision, but

never a loss of an order of magnitude. The other implicit numeric conversions never lose any information.

Also note that

- Any integral numeric type is implicitly convertible to any floating-point numeric type.
- There are no implicit conversions to the `byte` and `sbyte` types. There are no implicit conversions from the `double` and `decimal` types.
- There are no implicit conversions between the `decimal` type and the `float` or `double` types.
- A value of a constant expression of type `int` (for example, a value represented by an integer literal) can be implicitly converted to `sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, `nint`, or `nuint`, if it's within the range of the destination type:

  ```
  C#

  byte a = 13;
  byte b = 300;  // CS0031: Constant value '300' cannot be converted to a
  'byte'
  ```

  As the preceding example shows, if the constant value is not within the range of the destination type, a compiler error CS0031 occurs.

Explicit numeric conversions

The following table shows the predefined explicit conversions between the built-in numeric types for which there is no implicit conversion:

Expand table

From	To
sbyte	`byte`, `ushort`, `uint`, `ulong`, or `nuint`
byte	`sbyte`
short	`sbyte`, `byte`, `ushort`, `uint`, `ulong`, or `nuint`
ushort	`sbyte`, `byte`, or `short`

From	To
int	`sbyte`, `byte`, `short`, `ushort`, `uint`, `ulong`, or `nuint`
uint	`sbyte`, `byte`, `short`, `ushort`, `int`, or `nint`
long	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, `nint`, or `nuint`
ulong	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `nint`, or `nuint`
float	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `decimal`, `nint`, or `nuint`
double	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `decimal`, `nint`, or `nuint`
decimal	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `float`, `double`, `nint`, or `nuint`
nint	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `ulong`, or `nuint`
nuint	`sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, or `nint`

> **Note**
>
> An explicit numeric conversion might result in data loss or throw an exception, typically an **OverflowException**.

Also note that:

- When you convert a value of an integral type to another integral type, the result depends on the overflow-checking context. In a checked context, the conversion succeeds if the source value is within the range of the destination type. Otherwise, an OverflowException is thrown. In an unchecked context, the conversion always succeeds, and proceeds as follows:
  - If the source type is larger than the destination type, then the source value is truncated by discarding its "extra" most significant bits. The result is then treated as a value of the destination type.
  - If the source type is smaller than the destination type, then the source value is either sign-extended or zero-extended so that it's of the same size as the destination type. Sign-extension is used if the source type is signed; zero-extension is used if the source type is unsigned. The result is then treated as a value of the destination type.
  - If the source type is the same size as the destination type, then the source value is treated as a value of the destination type.

- When you convert a `decimal` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, an OverflowException is thrown.
- When you convert a `double` or `float` value to an integral type, this value is rounded towards zero to the nearest integral value. If the resulting integral value is outside the range of the destination type, the result depends on the overflow-checking context. In a checked context, an OverflowException is thrown, while in an unchecked context, the result is an unspecified value of the destination type.
- When you convert `double` to `float`, the `double` value is rounded to the nearest `float` value. If the `double` value is too small or too large to fit into the `float` type, the result is zero or infinity.
- When you convert `float` or `double` to `decimal`, the source value is converted to `decimal` representation and rounded to the nearest number after the 28th decimal place if necessary. Depending on the value of the source value, one of the following results may occur:
  - If the source value is too small to be represented as a `decimal`, the result becomes zero.
  - If the source value is NaN (not a number), infinity, or too large to be represented as a `decimal`, an OverflowException is thrown.
- When you convert `decimal` to `float` or `double`, the source value is rounded to the nearest `float` or `double` value, respectively.

C# language specification

For more information, see the following sections of the C# language specification:

- Implicit numeric conversions
- Explicit numeric conversions

See also

- Casting and type conversions

bool (C# reference)

Article • 01/25/2022

The `bool` type keyword is an alias for the .NET System.Boolean structure type that represents a Boolean value, which can be either `true` or `false`.

To perform logical operations with values of the `bool` type, use Boolean logical operators. The `bool` type is the result type of comparison and equality operators. A `bool` expression can be a controlling conditional expression in the if, do, while, and for statements and in the conditional operator ?:.

The default value of the `bool` type is `false`.

Literals

You can use the `true` and `false` literals to initialize a `bool` variable or to pass a `bool` value:

```
C#
```

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not
checked
```

Three-valued Boolean logic

Use the nullable `bool?` type, if you need to support the three-valued logic, for example, when you work with databases that support a three-valued Boolean type. For the `bool?` operands, the predefined `&` and `|` operators support the three-valued logic. For more information, see the Nullable Boolean logical operators section of the Boolean logical operators article.

For more information about nullable value types, see Nullable value types.

Conversions

C# provides only two conversions that involve the `bool` type. Those are an implicit conversion to the corresponding nullable `bool?` type and an explicit conversion from

the `bool?` type. However, .NET provides additional methods that you can use to convert to or from the `bool` type. For more information, see the Converting to and from Boolean values section of the System.Boolean API reference page.

C# language specification

For more information, see The bool type section of the C# language specification.

See also

- Value types
- true and false operators

char (C# reference)

Article • 02/28/2025

The `char` type keyword is an alias for the .NET System.Char structure type that represents a Unicode UTF-16 code unit, typically a UTF-16 character.

Expand table

Type	Range	Size	.NET type
`char`	U+0000 to U+FFFF	16 bit	System.Char

The default value of the `char` type is `\0`, that is, U+0000.

The `char` type supports comparison, equality, increment, and decrement operators. Moreover, for `char` operands, arithmetic and bitwise logical operators perform an operation on the corresponding code points and produce the result as an `int` value.

The string type represents text as a sequence of `char` values.

Literals

You can specify a `char` value with:

- a character literal.
- a Unicode escape sequence, which is `\u` followed by the four-symbol hexadecimal representation of a character code.
- a hexadecimal escape sequence, which is `\x` followed by the hexadecimal representation of a character code.

C#

```
var chars = new[]
{
    'j',
    '\u006A',
    '\x006A',
    (char)106,
};
Console.WriteLine(string.Join(" ", chars));  // output: j j j j
```

As the preceding example shows, you can also cast the value of a character code into the corresponding `char` value.

> **Note**
>
> In a Unicode escape sequence, you must specify all four hexadecimal digits. That is, `\u006A` is a valid escape sequence, while `\u06A` and `\u6A` are invalid.
>
> In a hexadecimal escape sequence, you can omit the leading zeros. That is, the `\x006A`, `\x06A`, and `\x6A` escape sequences are valid and correspond to the same character.

Conversions

The `char` type is implicitly convertible to the following integral types: `ushort`, `int`, `uint`, `long`, `ulong`, `nint`, and `nuint`. It's also implicitly convertible to the built-in floating-point numeric types: `float`, `double`, and `decimal`. It's explicitly convertible to `sbyte`, `byte`, and `short` integral types.

There are no implicit conversions from other types to the `char` type. However, any integral or floating-point numeric type is explicitly convertible to `char`.

C# language specification

For more information, see the Integral types section of the C# language specification.

See also

- Value types
- Strings
- System.Text.Rune
- Character encoding in .NET

Enumeration types (C# reference)

08/08/2025

An *enumeration type* (or *enum type*) is a value type defined by a set of named constants of the underlying integral numeric type. To define an enumeration type, use the `enum` keyword and specify the names of *enum members*:

```
C#

enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

By default, the associated constant values of enum members are of type `int`; they start with zero and increase by one following the definition text order. You can explicitly specify any other integral numeric type as an underlying type of an enumeration type. You can also explicitly specify the associated constant values, as the following example shows:

```
C#

enum ErrorCode : ushort
{
    None = 0,
    Unknown = 1,
    ConnectionLost = 100,
    OutlierReading = 200
}
```

You cannot define a method inside the definition of an enumeration type. To add functionality to an enumeration type, create an extension method.

The default value of an enumeration type `E` is the value produced by expression `(E)0`, even if zero doesn't have the corresponding enum member.

Implicit conversions from zero

C# allows implicit conversions from the literal value `0` to any enum type, and from `const` values equal to zero. This behavior can lead to unexpected results when an enum doesn't include a member with the value zero:

C#

```
public enum GpioPort
{
    GpioA = 1,
    GpioB,
    GpioC,
    GpioD
}

public class ZeroConversionExample
{
    public static void Main()
    {
        // This compiles without warning but creates an invalid enum value
        GpioPort port1 = (GpioPort)0;
        Console.WriteLine($"port1: {port1}"); // Output: port1: 0

        // This also compiles due to implicit conversion from zero
        GpioPort port2 = GetPort(0);
        Console.WriteLine($"port2: {port2}"); // Output: port2: 0

        // Check if the enum value is valid
        bool isValid1 = Enum.IsDefined(typeof(GpioPort), port1);
        bool isValid2 = Enum.IsDefined(typeof(GpioPort), port2);
        Console.WriteLine($"port1 is valid: {isValid1}"); // Output: port1 is
valid: False
        Console.WriteLine($"port2 is valid: {isValid2}"); // Output: port2 is
valid: False

        // Safer approach - validate enum values
        if (Enum.IsDefined(typeof(GpioPort), 0))
        {
            GpioPort safePort = (GpioPort)0;
        }
        else
        {
            Console.WriteLine("Value 0 is not a valid GpioPort");
            // Handle the invalid case appropriately
        }
    }

    public static GpioPort GetPort(GpioPort port)
    {
        return port;
    }
}
```

In the preceding example, both `port1` and `port2` are assigned the value `0`, but `GpioPort` has no member with that value. The Enum.IsDefined method confirms these are invalid enum values.

This implicit conversion exists because the 0 bit pattern is the default for all struct types, including all enum types. However, it can introduce bugs in your code. To avoid these issues:

- You should almost always define a member with value `0` in your enums.
- Use Enum.IsDefined to validate enum values when converting from numeric types.
- Be cautious when using numeric parameters that might be implicitly converted to enum types.

You use an enumeration type to represent a choice from a set of mutually exclusive values or a combination of choices. To represent a combination of choices, define an enumeration type as bit flags.

Enumeration types as bit flags

If you want an enumeration type to represent a combination of choices, define enum members for those choices such that an individual choice is a bit field. That is, the associated values of those enum members should be the powers of two. Then, you can use the bitwise logical operators | or & to combine choices or intersect combinations of choices, respectively. To indicate that an enumeration type declares bit fields, apply the Flags attribute to it. As the following example shows, you can also include some typical combinations in the definition of an enumeration type.

C#

```
[Flags]
public enum Days
{
    None      = 0b_0000_0000,  // 0
    Monday    = 0b_0000_0001,  // 1
    Tuesday   = 0b_0000_0010,  // 2
    Wednesday = 0b_0000_0100,  // 4
    Thursday  = 0b_0000_1000,  // 8
    Friday    = 0b_0001_0000,  // 16
    Saturday  = 0b_0010_0000,  // 32
    Sunday    = 0b_0100_0000,  // 64
    Weekend   = Saturday | Sunday
}

public class FlagsEnumExample
{
    public static void Main()
    {
        Days meetingDays = Days.Monday | Days.Wednesday | Days.Friday;
        Console.WriteLine(meetingDays);
        // Output:
        // Monday, Wednesday, Friday

        Days workingFromHomeDays = Days.Thursday | Days.Friday;
```

```
        Console.WriteLine($"Join a meeting by phone on {meetingDays &
workingFromHomeDays}");
        // Output:
        // Join a meeting by phone on Friday

        bool isMeetingOnTuesday = (meetingDays & Days.Tuesday) == Days.Tuesday;
        Console.WriteLine($"Is there a meeting on Tuesday: {isMeetingOnTuesday}");
        // Output:
        // Is there a meeting on Tuesday: False

        var a = (Days)37;
        Console.WriteLine(a);
        // Output:
        // Monday, Wednesday, Saturday
    }
}
```

For more information and examples, see the System.FlagsAttribute API reference page and the Non-exclusive members and the Flags attribute section of the System.Enum API reference page.

The System.Enum type and enum constraint

The System.Enum type is the abstract base class of all enumeration types. It provides a number of methods to get information about an enumeration type and its values. For more information and examples, see the System.Enum API reference page.

You can use `System.Enum` in a base class constraint (that is known as the enum constraint) to specify that a type parameter is an enumeration type. Any enumeration type also satisfies the `struct` constraint, which is used to specify that a type parameter is a non-nullable value type.

Conversions

For any enumeration type, there exist explicit conversions between the enumeration type and its underlying integral type. If you cast an enum value to its underlying type, the result is the associated integral value of an enum member.

C#

```
public enum Season
{
    Spring,
    Summer,
    Autumn,
    Winter
}
```

```
public class EnumConversionExample
{
    public static void Main()
    {
        Season a = Season.Autumn;
        Console.WriteLine($"Integral value of {a} is {(int)a}");  // output:
Integral value of Autumn is 2

        var b = (Season)1;
        Console.WriteLine(b);  // output: Summer

        var c = (Season)4;
        Console.WriteLine(c);  // output: 4
    }
}
```

Use the Enum.IsDefined method to determine whether an enumeration type contains an enum member with the certain associated value.

For any enumeration type, there exist boxing and unboxing conversions to and from the System.Enum type, respectively.

C# language specification

For more information, see the following sections of the C# language specification:

- Enums
- Enum values and operations
- Enumeration logical operators
- Enumeration comparison operators
- Explicit enumeration conversions
- Implicit enumeration conversions

See also

- Enumeration format strings
- Design guidelines - Enum design
- Design guidelines - Enum naming conventions
- switch expression
- switch statement

Structure types (C# reference)

Article • 12/04/2024

A *structure type* (or *struct type*) is a value type that can encapsulate data and related functionality. You use the `struct` keyword to define a structure type:

C#

```
public struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; }
    public double Y { get; }

    public override string ToString() => $"({X}, {Y})";
}
```

For information about `ref struct` and `readonly ref struct` types, see the ref structure types article.

Structure types have *value semantics*. That is, a variable of a structure type contains an instance of the type. By default, variable values are copied on assignment, passing an argument to a method, and returning a method result. For structure-type variables, an instance of the type is copied. For more information, see Value types.

Typically, you use structure types to design small data-centric types that provide little or no behavior. For example, .NET uses structure types to represent a number (both integer and real), a Boolean value, a Unicode character, a time instance. If you're focused on the behavior of a type, consider defining a class. Class types have *reference semantics*. That is, a variable of a class type contains a reference to an instance of the type, not the instance itself.

Because structure types have value semantics, we recommend you define *immutable* structure types.

`readonly` struct

You use the `readonly` modifier to declare that a structure type is immutable. All data members of a `readonly` struct must be read-only as follows:

- Any field declaration must have the readonly modifier.
- Any property, including automatically implemented ones, must be read-only or init only. Init-only setters are only available from C# version 9 onwards.

That guarantees that no member of a `readonly` struct modifies the state of the struct. That means that other instance members except constructors are implicitly readonly.

> **Note**
>
> In a `readonly` struct, a data member of a mutable reference type still can mutate its own state. For example, you can't replace a List<T> instance, but you can add new elements to it.

The following code defines a `readonly` struct with init-only property setters:

```
C#

public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}
```

`readonly` instance members

You can also use the `readonly` modifier to declare that an instance member doesn't modify the state of a struct. If you can't declare the whole structure type as `readonly`, use the `readonly` modifier to mark the instance members that don't modify the state of the struct.

Within a `readonly` instance member, you can't assign to structure's instance fields. However, a `readonly` member can call a non-`readonly` member. In that case, the

compiler creates a copy of the structure instance and calls the non-`readonly` member on that copy. As a result, the original structure instance isn't modified.

Typically, you apply the `readonly` modifier to the following kinds of instance members:

- methods:

  ```
  C#

  public readonly double Sum()
  {
      return X + Y;
  }
  ```

  You can also apply the `readonly` modifier to methods that override methods declared in System.Object:

  ```
  C#

  public readonly override string ToString() => $"({X}, {Y})";
  ```

- properties and indexers:

  ```
  C#

  private int counter;
  public int Counter
  {
      readonly get => counter;
      set => counter = value;
  }
  ```

  If you need to apply the `readonly` modifier to both accessors of a property or indexer, apply it in the declaration of the property or indexer.

  > **Note**
  >
  > The compiler declares a `get` accessor of an automatically implemented property as `readonly`, regardless of presence of the `readonly` modifier in a property declaration.

  You can apply the `readonly` modifier to a property or indexer with an `init` accessor:

```
C#
```

```
public readonly double X { get; init; }
```

You can apply the `readonly` modifier to static fields of a structure type, but not any other static members, such as properties or methods.

The compiler can make use of the `readonly` modifier for performance optimizations. For more information, see Avoiding allocations.

Nondestructive mutation

You can use the with expression to produce a copy of a structure-type instance with the specified properties and fields modified. You use object initializer syntax to specify what members to modify and their new values, as the following example shows:

```
C#
```

```
public readonly struct Coords
{
    public Coords(double x, double y)
    {
        X = x;
        Y = y;
    }

    public double X { get; init; }
    public double Y { get; init; }

    public override string ToString() => $"({X}, {Y})";
}

public static void Main()
{
    var p1 = new Coords(0, 0);
    Console.WriteLine(p1);  // output: (0, 0)

    var p2 = p1 with { X = 3 };
    Console.WriteLine(p2);  // output: (3, 0)

    var p3 = p1 with { X = 1, Y = 4 };
    Console.WriteLine(p3);  // output: (1, 4)
}
```

`record` struct

You can define record structure types. Record types provide built-in functionality for encapsulating data. You can define both `record struct` and `readonly record struct` types. A record struct can't be a ref struct. For more information and examples, see Records.

Inline arrays

Beginning with C# 12, you can declare *inline arrays* as a `struct` type:

```
C#
```

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBuffer
{
    private char _firstElement;
}
```

An inline array is a structure that contains a contiguous block of N elements of the same type. It's a safe-code equivalent of the fixed buffer declaration available only in unsafe code. An inline array is a `struct` with the following characteristics:

- It contains a single field.
- The struct doesn't specify an explicit layout.

In addition, the compiler validates the System.Runtime.CompilerServices.InlineArrayAttribute attribute:

- The length must be greater than zero (`> 0`).
- The target type must be a struct.

In most cases, an inline array can be accessed like an array, both to read and write values. In addition, you can use the range and index operators.

There are minimal restrictions on the type of the single field of an inline array. It can't be a pointer type:

```
C#
```

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBufferWithPointer
{
    private unsafe char* _pointerElement;    // CS9184
}
```

But it can be any reference type, or any value type:

C#

```
[System.Runtime.CompilerServices.InlineArray(10)]
public struct CharBufferWithReferenceType
{
    private string _referenceElement;
}
```

You can use inline arrays with almost any C# data structure.

Inline arrays are an advanced language feature. They're intended for high-performance scenarios where an inline, contiguous block of elements is faster than other alternative data structures. You can learn more about inline arrays from the feature speclet.

Struct initialization and default values

A variable of a `struct` type directly contains the data for that `struct`. That creates a distinction between an uninitialized `struct`, which has its default value and an initialized `struct`, which stores values set by constructing it. For example consider the following code:

C#

```
public readonly struct Measurement
{
    public Measurement()
    {
        Value = double.NaN;
        Description = "Undefined";
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; }

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement();
    Console.WriteLine(m1);  // output: NaN (Undefined)

    var m2 = default(Measurement);
```

```
    Console.WriteLine(m2);  // output: 0 ()

    var ms = new Measurement[2];
    Console.WriteLine(string.Join(", ", ms));  // output: 0 (), 0 ()
}
```

As the preceding example shows, the default value expression ignores a parameterless constructor and produces the default value of the structure type. Structure-type array instantiation also ignores a parameterless constructor and produces an array populated with the default values of a structure type.

The most common situation where you see default values is in arrays or in other collections where internal storage includes blocks of variables. The following example creates an array of 30 `TemperatureRange` structures, each of which has the default value:

C#

```
// All elements have default values of 0:
TemperatureRange[] lastMonth = new TemperatureRange[30];
```

All of a struct's member fields must be *definitely assigned* when created because `struct` types directly store their data. The `default` value of a struct *definitely assigned* all fields to 0. All fields must be definitely assigned when a constructor is invoked. You initialize fields using the following mechanisms:

- You can add *field initializers* to any field or auto implemented property.
- You can initialize any fields, or auto properties, in the body of the constructor.

Beginning with C# 11, if you don't initialize all fields in a struct, the compiler adds code to the constructor that initializes those fields to the default value. A struct assigned to its `default` value is initialized to the 0-bit pattern. A struct initialized with `new` is initialized to the 0-bit pattern, followed by executing any field initializers and a constructor.

C#

```
public readonly struct Measurement
{
    public Measurement(double value)
    {
        Value = value;
    }

    public Measurement(double value, string description)
    {
        Value = value;
        Description = description;
    }
```

```
    public Measurement(string description)
    {
        Description = description;
    }

    public double Value { get; init; }
    public string Description { get; init; } = "Ordinary measurement";

    public override string ToString() => $"{Value} ({Description})";
}

public static void Main()
{
    var m1 = new Measurement(5);
    Console.WriteLine(m1);  // output: 5 (Ordinary measurement)

    var m2 = new Measurement();
    Console.WriteLine(m2);  // output: 0 ()

    var m3 = default(Measurement);
    Console.WriteLine(m3);  // output: 0 ()
}
```

Every `struct` has a `public` parameterless constructor. If you write a parameterless constructor, it must be public. If a struct declares any field initializers, it must explicitly declare a constructor. That constructor need not be parameterless. If a struct declares a field initializer but no constructors, the compiler reports an error. Any explicitly declared constructor (with parameters, or parameterless) executes all field initializers for that struct. All fields without a field initializer or an assignment in a constructor are set to the default value. For more information, see the Parameterless struct constructors feature proposal note.

Beginning with C# 12, `struct` types can define a primary constructor as part of its declaration. Primary constructors provide a concise syntax for constructor parameters that can be used throughout the `struct` body, in any member declaration for that struct.

If all instance fields of a structure type are accessible, you can also instantiate it without the `new` operator. In that case you must initialize all instance fields before the first use of the instance. The following example shows how to do that:

C#

```
public static class StructWithoutNew
{
    public struct Coords
    {
```

```
        public double x;
        public double y;
    }

    public static void Main()
    {
        Coords p;
        p.x = 3;
        p.y = 4;
        Console.WriteLine($"({p.x}, {p.y})");  // output: (3, 4)
    }
}
```

For the built-in value types, use the corresponding literals to specify a value of the type.

Limitations with the design of a structure type

Structs have most of the capabilities of a class type. There are some exceptions:

- A structure type can't inherit from other class or structure type and it can't be the base of a class. However, a structure type can implement interfaces.
- You can't declare a finalizer within a structure type.
- Before C# 11, a constructor of a structure type must initialize all instance fields of the type.

Passing structure-type variables by reference

When you pass a structure-type variable to a method as an argument or return a structure-type value from a method, the whole instance of a structure type is copied. Pass by value can affect the performance of your code in high-performance scenarios that involve large structure types. You can avoid value copying by passing a structure-type variable by reference. Use the `ref`, `out`, `in`, or `ref readonly` method parameter modifiers to indicate that an argument must be passed by reference. Use ref returns to return a method result by reference. For more information, see Avoid allocations.

`struct` constraint

You also use the `struct` keyword in the struct constraint to specify that a type parameter is a non-nullable value type. Both structure and enumeration types satisfy the `struct` constraint.

Conversions

For any structure type (except ref struct types), there exist boxing and unboxing conversions to and from the System.ValueType and System.Object types. There exist also boxing and unboxing conversions between a structure type and any interface that it implements.

C# language specification

For more information, see the Structs section of the C# language specification.

For more information about `struct` features, see the following feature proposal notes:

- Readonly structs
- Readonly instance members
- Parameterless struct constructors
- Allow with expression on structs
- Record structs
- Auto default structs

See also

- The C# type system
- Design guidelines - Choosing between class and struct
- Design guidelines - Struct design
- Resolve errors and warnings with inline array declarations

`ref` structure types (C# reference)

Article • 01/29/2025

You can use the `ref` modifier in the declaration of a structure type. Instances of a `ref struct` type are allocated on the stack and can't escape to the managed heap. To ensure that, the compiler limits the usage of `ref struct` types as follows:

- A `ref struct` can't be the element type of an array.
- A `ref struct` can't be a declared type of a field of a class or a non-`ref struct`.
- A `ref struct` can't be boxed to System.ValueType or System.Object.
- A `ref struct` variable can't be captured in a lambda expression or a local function.
- Before C# 13, `ref struct` variables can't be used in an `async` method. Beginning with C# 13, a `ref struct` variable can't be used in the same block as the await expression in an async method. However, you can use `ref struct` variables in synchronous methods, for example, in methods that return Task or Task<TResult>.
- Before C# 13, a `ref struct` variable can't be used in iterators. Beginning with C# 13, `ref struct` types and `ref` locals can be used in iterators, provided they aren't in code segments with the `yield return` statement.
- Before C# 13, a `ref struct` can't implement interfaces. Beginning with C# 13, a `ref` struct can implement interfaces, but must adhere to the ref safety rules. For example, a `ref struct` type can't be converted to the interface type because that requires a boxing conversion.
- Before C# 13, a `ref struct` can't be a type argument. Beginning with C# 13, a `ref struct` can be the type argument when the type parameter specifies the `allows ref struct` in its `where` clause.

Typically, you define a `ref struct` type when you need a type that also includes data members of `ref struct` types:

C#

```
public ref struct CustomRef
{
    public bool IsValid;
    public Span<int> Inputs;
    public Span<int> Outputs;
}
```

To declare a `ref struct` as `readonly`, combine the `readonly` and `ref` modifiers in the type declaration (the `readonly` modifier must come before the `ref` modifier):

C#

```
public readonly ref struct ConversionRequest
{
    public ConversionRequest(double rate, ReadOnlySpan<double> values)
    {
        Rate = rate;
        Values = values;
    }

    public double Rate { get; }
    public ReadOnlySpan<double> Values { get; }
}
```

In .NET, examples of a `ref struct` are System.Span<T> and System.ReadOnlySpan<T>.

`ref` fields

Beginning with C# 11, you can declare a `ref` field in a `ref struct`, as the following example shows:

C#

```
public ref struct RefFieldExample
{
    private ref int number;

    public int GetNumber()
    {
        if (System.Runtime.CompilerServices.Unsafe.IsNullRef(ref number))
        {
            throw new InvalidOperationException("The number ref field is not
initialized.");
        }

        return number;
    }
}
```

A `ref` field can have the `null` value. Use the Unsafe.IsNullRef<T>(T) method to determine if a `ref` field is `null`.

You can apply the `readonly` modifier to a `ref` field in the following ways:

- `readonly ref`: You can ref reassign such a field with the `= ref` operator only inside a constructor or an init accessor. You can assign a value with the `=` operator at any point allowed by the field access modifier.

- `ref readonly`: At any point, you can't assign a value with the `=` operator to such a field. However, you can ref reassign a field with the `= ref` operator.
- `readonly ref readonly`: You can only ref reassign such a field in a constructor or an `init` accessor. At any point, you can't assign a value to the field.

The compiler ensures that a reference stored in a `ref` field doesn't outlive its referent.

The `ref` fields feature enables a safe implementation of types like System.Span<T>:

```
C#
```

```
public readonly ref struct Span<T>
{
    internal readonly ref T _reference;
    private readonly int _length;

    // Omitted for brevity...
}
```

The `Span<T>` type stores a reference through which it accesses the contiguous elements in memory. The use of a reference enables a `Span<T>` instance to avoid copying the storage it refers to.

The disposable pattern

You can define a disposable `ref struct`. To do that, ensure that a `ref struct` fits the disposable pattern. That is, it has an instance `Dispose` method, which is accessible, parameterless and has a `void` return type. You can use the using statement or declaration with an instance of a disposable `ref struct`.

Beginning with C# 13, you can also implement the IDisposable on `ref struct` types. However, overload resolution prefers the disposable pattern to the interface method. The compiler resolves to an `IDisposable.Dispose` method only when a suitable `Dispose` method isn't found.

Restrictions for `ref struct` types that implement an interface

These restrictions ensure that a `ref struct` type that implements an interface obeys the necessary ref safety rules.

- A `ref struct` can't be converted to an instance of an interface it implements. This restriction includes the implicit conversion when you use a `ref struct` type as an argument when the parameter is an interface type. The conversion results in a boxing conversion, which violates ref safety. A `ref struct` can declare methods as explicit interface declarations. However, those methods can be accessed only from generic methods where the type parameter allows ref struct types.
- A `ref struct` that implements an interface *must* implement all instance interface members. The `ref struct` must implement instance members even when the interface includes a default implementation.

The compiler enforces these restrictions. If you write `ref struct` types that implement interfaces, each new update might include new default interface members. Until you provide an implementation for any new instance methods, your application won't compile. You can't provide a specific implementation for a `static` interface method with a default implementation.

> **ⓘ Important**
>
> A `ref struct` that implements an interface includes the potential for later source-breaking and binary-breaking changes. The break occurs if a `ref struct` implements an interface defined in another assembly, and that assembly provides an update which adds default members to that interface.
>
> The source-break happens when you recompile the `ref struct`: It must implement the new member, even though there's a default implementation.
>
> The binary-break happens if you upgrade the external assembly without recompiling the `ref struct` type *and* the updated code calls the default implementation of the new method. The runtime throws an exception when the default member is accessed.

C# language specification

For more information, see the following sections of the C# language specification:

- Structs: Ref modifier
- Safe context constraint for ref struct types

For more information about `ref` fields, see the Low-level struct improvements proposal note.

See also

- The C# type system

Tuple types (C# reference)

Article • 05/17/2023

The *tuples* feature provides concise syntax to group multiple data elements in a lightweight data structure. The following example shows how you can declare a tuple variable, initialize it, and access its data members:

```
C#

(double, int) t1 = (4.5, 3);
Console.WriteLine($"Tuple with elements {t1.Item1} and {t1.Item2}.");
// Output:
// Tuple with elements 4.5 and 3.

(double Sum, int Count) t2 = (4.5, 3);
Console.WriteLine($"Sum of {t2.Count} elements is {t2.Sum}.");
// Output:
// Sum of 3 elements is 4.5.
```

As the preceding example shows, to define a tuple type, you specify types of all its data members and, optionally, the field names. You can't define methods in a tuple type, but you can use the methods provided by .NET, as the following example shows:

```
C#

(double, int) t = (4.5, 3);
Console.WriteLine(t.ToString());
Console.WriteLine($"Hash code of {t} is {t.GetHashCode()}.");
// Output:
// (4.5, 3)
// Hash code of (4.5, 3) is 718460086.
```

Tuple types support equality operators `==` and `!=`. For more information, see the Tuple equality section.

Tuple types are value types; tuple elements are public fields. That makes tuples *mutable* value types.

You can define tuples with an arbitrary large number of elements:

```
C#

var t =
(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);
Console.WriteLine(t.Item26);  // output: 26
```

Use cases of tuples

One of the most common use cases of tuples is as a method return type. That is, instead of defining out method parameters, you can group method results in a tuple return type, as the following example shows:

C#

```
int[] xs = new int[] { 4, 7, 9 };
var limits = FindMinMax(xs);
Console.WriteLine($"Limits of [{string.Join(" ", xs)}] are {limits.min} and
{limits.max}");
// Output:
// Limits of [4 7 9] are 4 and 9

int[] ys = new int[] { -9, 0, 67, 100 };
var (minimum, maximum) = FindMinMax(ys);
Console.WriteLine($"Limits of [{string.Join(" ", ys)}] are {minimum} and
{maximum}");
// Output:
// Limits of [-9 0 67 100] are -9 and 100

(int min, int max) FindMinMax(int[] input)
{
    if (input is null || input.Length == 0)
    {
        throw new ArgumentException("Cannot find minimum and maximum of a
null or empty array.");
    }

    // Initialize min to MaxValue so every value in the input
    // is less than this initial value.
    var min = int.MaxValue;
    // Initialize max to MinValue so every value in the input
    // is greater than this initial value.
    var max = int.MinValue;
    foreach (var i in input)
    {
        if (i < min)
        {
            min = i;
        }
        if (i > max)
        {
            max = i;
        }
    }
```

```
    return (min, max);
}
```

As the preceding example shows, you can work with the returned tuple instance directly or deconstruct it in separate variables.

You can also use tuple types instead of anonymous types; for example, in LINQ queries. For more information, see Choosing between anonymous and tuple types.

Typically, you use tuples to group loosely related data elements. In public APIs, consider defining a class or a structure type.

Tuple field names

You explicitly specify tuple fields names in a tuple initialization expression or in the definition of a tuple type, as the following example shows:

C#

```
var t = (Sum: 4.5, Count: 3);
Console.WriteLine($"Sum of {t.Count} elements is {t.Sum}.");

(double Sum, int Count) d = (4.5, 3);
Console.WriteLine($"Sum of {d.Count} elements is {d.Sum}.");
```

If you don't specify a field name, it may be inferred from the name of the corresponding variable in a tuple initialization expression, as the following example shows:

C#

```
var sum = 4.5;
var count = 3;
var t = (sum, count);
Console.WriteLine($"Sum of {t.count} elements is {t.sum}.");
```

That's called tuple projection initializers. The name of a variable isn't projected onto a tuple field name in the following cases:

- The candidate name is a member name of a tuple type, for example, `Item3`, `ToString`, or `Rest`.
- The candidate name is a duplicate of another tuple field name, either explicit or implicit.

In the preceding cases, you either explicitly specify the name of a field or access a field by its default name.

The default names of tuple fields are `Item1`, `Item2`, `Item3` and so on. You can always use the default name of a field, even when a field name is specified explicitly or inferred, as the following example shows:

C#

```
var a = 1;
var t = (a, b: 2, 3);
Console.WriteLine($"The 1st element is {t.Item1} (same as {t.a}).");
Console.WriteLine($"The 2nd element is {t.Item2} (same as {t.b}).");
Console.WriteLine($"The 3rd element is {t.Item3}.");
// Output:
// The 1st element is 1 (same as 1).
// The 2nd element is 2 (same as 2).
// The 3rd element is 3.
```

Tuple assignment and tuple equality comparisons don't take field names into account.

At compile time, the compiler replaces non-default field names with the corresponding default names. As a result, explicitly specified or inferred field names aren't available at run time.

> **Tip**
>
> Enable .NET code style rule **IDE0037** to set a preference on inferred or explicit tuple field names.

Beginning with C# 12, you can specify an alias for a tuple type with a using directive. The following example adds a `global using` alias for a tuple type with two integer values for an allowed `Min` and `Max` value:

C#

```
global using BandPass = (int Min, int Max);
```

After declaring the alias, you can use the `BandPass` name as an alias for that tuple type:

C#

```
BandPass bracket = (40, 100);
Console.WriteLine($"The bandpass filter is {bracket.Min} to {bracket.Max}");
```

An alias doesn't introduce a new *type*, but only creates a synonym for an existing type. You can deconstruct a tuple declared with the `BandPass` alias the same as you can with its underlying tuple type:

```
C#

(int a , int b) = bracket;
Console.WriteLine($"The bracket is {a} to {b}");
```

As with tuple assignment or deconstruction, the tuple member names don't need to match; the types do.

Similarly, a second alias with the same arity and member types can be used interchangeably with the original alias. You could declare a second alias:

```
C#

using Range = (int Minimum, int Maximum);
```

You can assign a `Range` tuple to a `BandPass` tuple. As with all tuple assignment, the field names need not match, only the types and the arity.

```
C#

Range r = bracket;
Console.WriteLine($"The range is {r.Minimum} to {r.Maximum}");
```

An alias for a tuple type provides more semantic information when you use tuples. It doesn't introduce a new type. To provide type safety, you should declare a positional record instead.

Tuple assignment and deconstruction

C# supports assignment between tuple types that satisfy both of the following conditions:

- both tuple types have the same number of elements
- for each tuple position, the type of the right-hand tuple element is the same as or implicitly convertible to the type of the corresponding left-hand tuple element

Tuple element values are assigned following the order of tuple elements. The names of tuple fields are ignored and not assigned, as the following example shows:

C#

```
(int, double) t1 = (17, 3.14);
(double First, double Second) t2 = (0.0, 1.0);
t2 = t1;
Console.WriteLine($"{nameof(t2)}: {t2.First} and {t2.Second}");
// Output:
// t2: 17 and 3.14

(double A, double B) t3 = (2.0, 3.0);
t3 = t2;
Console.WriteLine($"{nameof(t3)}: {t3.A} and {t3.B}");
// Output:
// t3: 17 and 3.14
```

You can also use the assignment operator `=` to *deconstruct* a tuple instance in separate variables. You can do that in many ways:

- Use the `var` keyword outside the parentheses to declare implicitly typed variables and let the compiler infer their types:

  C#

  ```
  var t = ("post office", 3.6);
  var (destination, distance) = t;
  Console.WriteLine($"Distance to {destination} is {distance}
  kilometers.");
  // Output:
  // Distance to post office is 3.6 kilometers.
  ```

- Explicitly declare the type of each variable inside parentheses:

  C#

  ```
  var t = ("post office", 3.6);
  (string destination, double distance) = t;
  Console.WriteLine($"Distance to {destination} is {distance}
  kilometers.");
  // Output:
  // Distance to post office is 3.6 kilometers.
  ```

- Declare some types explicitly and other types implicitly (with `var`) inside the parentheses:

  C#

  ```
  var t = ("post office", 3.6);
  (var destination, double distance) = t;
  ```

```
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

- Use existing variables:

```
C#

var destination = string.Empty;
var distance = 0.0;

var t = ("post office", 3.6);
(destination, distance) = t;
Console.WriteLine($"Distance to {destination} is {distance}
kilometers.");
// Output:
// Distance to post office is 3.6 kilometers.
```

The destination of a deconstruct expression can include both existing variables and variables declared in the deconstruction declaration.

You can also combine deconstruction with pattern matching to inspect the characteristics of fields in a tuple. The following example loops through several integers and prints those that are divisible by 3. It deconstructs the tuple result of Int32.DivRem and matches against a `Remainder` of 0:

```
C#

for (int i = 4; i < 20;  i++)
{
    if (Math.DivRem(i, 3) is ( Quotient: var q, Remainder: 0 ))
    {
        Console.WriteLine($"{i} is divisible by 3, with quotient {q}");
    }
}
```

For more information about deconstruction of tuples and other types, see Deconstructing tuples and other types.

Tuple equality

Tuple types support the `==` and `!=` operators. These operators compare members of the left-hand operand with the corresponding members of the right-hand operand following the order of tuple elements.

C#

```
(int a, byte b) left = (5, 10);
(long a, int b) right = (5, 10);
Console.WriteLine(left == right);  // output: True
Console.WriteLine(left != right);  // output: False

var t1 = (A: 5, B: 10);
var t2 = (B: 5, A: 10);
Console.WriteLine(t1 == t2);  // output: True
Console.WriteLine(t1 != t2);  // output: False
```

As the preceding example shows, the `==` and `!=` operations don't take into account tuple field names.

Two tuples are comparable when both of the following conditions are satisfied:

- Both tuples have the same number of elements. For example, `t1 != t2` doesn't compile if `t1` and `t2` have different numbers of elements.
- For each tuple position, the corresponding elements from the left-hand and right-hand tuple operands are comparable with the `==` and `!=` operators. For example, `(1, (2, 3)) == ((1, 2), 3)` doesn't compile because `1` isn't comparable with `(1, 2)`.

The `==` and `!=` operators compare tuples in short-circuiting way. That is, an operation stops as soon as it meets a pair of non equal elements or reaches the ends of tuples. However, before any comparison, *all* tuple elements are evaluated, as the following example shows:

C#

```
Console.WriteLine((Display(1), Display(2)) == (Display(3), Display(4)));

int Display(int s)
{
    Console.WriteLine(s);
    return s;
}
// Output:
// 1
// 2
// 3
// 4
// False
```

Tuples as out parameters

Typically, you refactor a method that has out parameters into a method that returns a tuple. However, there are cases in which an `out` parameter can be of a tuple type. The following example shows how to work with tuples as `out` parameters:

C#

```
var limitsLookup = new Dictionary<int, (int Min, int Max)>()
{
    [2] = (4, 10),
    [4] = (10, 20),
    [6] = (0, 23)
};

if (limitsLookup.TryGetValue(4, out (int Min, int Max) limits))
{
    Console.WriteLine($"Found limits: min is {limits.Min}, max is
{limits.Max}");
}
// Output:
// Found limits: min is 10, max is 20
```

Tuples vs `System.Tuple`

C# tuples, which are backed by System.ValueTuple types, are different from tuples that are represented by System.Tuple types. The main differences are as follows:

- `System.ValueTuple` types are value types. `System.Tuple` types are reference types.
- `System.ValueTuple` types are mutable. `System.Tuple` types are immutable.
- Data members of `System.ValueTuple` types are fields. Data members of `System.Tuple` types are properties.

C# language specification

For more information, see:

- Tuple types
- Tuple equality operators

See also

- Value types
- Choosing between anonymous and tuple types
- System.ValueTuple

Nullable value types (C# reference)

Article • 04/08/2023

A *nullable value type* `T?` represents all values of its underlying value type `T` and an additional null value. For example, you can assign any of the following three values to a `bool?` variable: `true`, `false`, or `null`. An underlying value type `T` cannot be a nullable value type itself.

Any nullable value type is an instance of the generic System.Nullable<T> structure. You can refer to a nullable value type with an underlying type `T` in any of the following interchangeable forms: `Nullable<T>` or `T?`.

You typically use a nullable value type when you need to represent the undefined value of an underlying value type. For example, a Boolean, or `bool`, variable can only be either `true` or `false`. However, in some applications a variable value can be undefined or missing. For example, a database field may contain `true` or `false`, or it may contain no value at all, that is, `NULL`. You can use the `bool?` type in that scenario.

Declaration and assignment

As a value type is implicitly convertible to the corresponding nullable value type, you can assign a value to a variable of a nullable value type as you would do that for its underlying value type. You can also assign the `null` value. For example:

C#

```
double? pi = 3.14;
char? letter = 'a';

int m2 = 10;
int? m = m2;

bool? flag = null;

// An array of a nullable value type:
int?[] arr = new int?[10];
```

The default value of a nullable value type represents `null`, that is, it's an instance whose Nullable<T>.HasValue property returns `false`.

Examination of an instance of a nullable value type

You can use the is operator with a type pattern to both examine an instance of a nullable value type for `null` and retrieve a value of an underlying type:

```
C#
```

```
int? a = 42;
if (a is int valueOfA)
{
    Console.WriteLine($"a is {valueOfA}");
}
else
{
    Console.WriteLine("a does not have a value");
}
// Output:
// a is 42
```

You always can use the following read-only properties to examine and get a value of a nullable value type variable:

- Nullable<T>.HasValue indicates whether an instance of a nullable value type has a value of its underlying type.
- Nullable<T>.Value gets the value of an underlying type if HasValue is `true`. If HasValue is `false`, the Value property throws an InvalidOperationException.

The following example uses the `HasValue` property to test whether the variable contains a value before displaying it:

```
C#
```

```
int? b = 10;
if (b.HasValue)
{
    Console.WriteLine($"b is {b.Value}");
}
else
{
    Console.WriteLine("b does not have a value");
}
// Output:
// b is 10
```

You can also compare a variable of a nullable value type with `null` instead of using the `HasValue` property, as the following example shows:

```
C#
```

```
int? c = 7;
if (c != null)
{
    Console.WriteLine($"c is {c.Value}");
}
else
{
    Console.WriteLine("c does not have a value");
}
// Output:
// c is 7
```

Conversion from a nullable value type to an underlying type

If you want to assign a value of a nullable value type to a non-nullable value type variable, you might need to specify the value to be assigned in place of `null`. Use the null-coalescing operator ?? to do that (you can also use the Nullable<T>.GetValueOrDefault(T) method for the same purpose):

```
C#
```

```
int? a = 28;
int b = a ?? -1;
Console.WriteLine($"b is {b}");  // output: b is 28

int? c = null;
int d = c ?? -1;
Console.WriteLine($"d is {d}");  // output: d is -1
```

If you want to use the default value of the underlying value type in place of `null`, use the Nullable<T>.GetValueOrDefault() method.

You can also explicitly cast a nullable value type to a non-nullable type, as the following example shows:

```
C#
```

```
int? n = null;
```

```
//int m1 = n;    // Doesn't compile
int n2 = (int)n; // Compiles, but throws an exception if n is null
```

At run time, if the value of a nullable value type is `null`, the explicit cast throws an InvalidOperationException.

A non-nullable value type `T` is implicitly convertible to the corresponding nullable value type `T?`.

Lifted operators

The predefined unary and binary operators or any overloaded operators that are supported by a value type `T` are also supported by the corresponding nullable value type `T?`. These operators, also known as *lifted operators*, produce `null` if one or both operands are `null`; otherwise, the operator uses the contained values of its operands to calculate the result. For example:

C#

```
int? a = 10;
int? b = null;
int? c = 10;

a++;        // a is 11
a = a * c;  // a is 110
a = a + b;  // a is null
```

> **Note**
>
> For the `bool?` type, the predefined `&` and `|` operators don't follow the rules described in this section: the result of an operator evaluation can be non-null even if one of the operands is `null`. For more information, see the Nullable Boolean logical operators section of the Boolean logical operators article.

For the comparison operators `<`, `>`, `<=`, and `>=`, if one or both operands are `null`, the result is `false`; otherwise, the contained values of operands are compared. Do not assume that because a particular comparison (for example, `<=`) returns `false`, the opposite comparison (`>`) returns `true`. The following example shows that 10 is

- neither greater than or equal to `null`
- nor less than `null`

C#

```
int? a = 10;
Console.WriteLine($"{a} >= null is {a >= null}");
Console.WriteLine($"{a} < null is {a < null}");
Console.WriteLine($"{a} == null is {a == null}");
// Output:
// 10 >= null is False
// 10 < null is False
// 10 == null is False

int? b = null;
int? c = null;
Console.WriteLine($"null >= null is {b >= c}");
Console.WriteLine($"null == null is {b == c}");
// Output:
// null >= null is False
// null == null is True
```

For the equality operator `==`, if both operands are `null`, the result is `true`, if only one of the operands is `null`, the result is `false`; otherwise, the contained values of operands are compared.

For the inequality operator `!=`, if both operands are `null`, the result is `false`, if only one of the operands is `null`, the result is `true`; otherwise, the contained values of operands are compared.

If there exists a user-defined conversion between two value types, the same conversion can also be used between the corresponding nullable value types.

Boxing and unboxing

An instance of a nullable value type `T?` is boxed as follows:

- If HasValue returns `false`, the null reference is produced.
- If HasValue returns `true`, the corresponding value of the underlying value type `T` is boxed, not the instance of Nullable<T>.

You can unbox a boxed value of a value type `T` to the corresponding nullable value type `T?`, as the following example shows:

C#

```
int a = 41;
object aBoxed = a;
int? aNullable = (int?)aBoxed;
```

```
Console.WriteLine($"Value of aNullable: {aNullable}");

object aNullableBoxed = aNullable;
if (aNullableBoxed is int valueOfA)
{
    Console.WriteLine($"aNullableBoxed is boxed int: {valueOfA}");
}
// Output:
// Value of aNullable: 41
// aNullableBoxed is boxed int: 41
```

How to identify a nullable value type

The following example shows how to determine whether a System.Type instance represents a constructed nullable value type, that is, the System.Nullable<T> type with a specified type parameter `T`:

C#

```
Console.WriteLine($"int? is {(IsNullable(typeof(int?)) ? "nullable" : "non
nullable")} value type");
Console.WriteLine($"int is {(IsNullable(typeof(int)) ? "nullable" : "non-
nullable")} value type");

bool IsNullable(Type type) => Nullable.GetUnderlyingType(type) != null;

// Output:
// int? is nullable value type
// int is non-nullable value type
```

As the example shows, you use the typeof operator to create a System.Type instance.

If you want to determine whether an instance is of a nullable value type, don't use the Object.GetType method to get a Type instance to be tested with the preceding code. When you call the Object.GetType method on an instance of a nullable value type, the instance is boxed to Object. As boxing of a non-null instance of a nullable value type is equivalent to boxing of a value of the underlying type, GetType returns a Type instance that represents the underlying type of a nullable value type:

C#

```
int? a = 17;
Type typeOfA = a.GetType();
Console.WriteLine(typeOfA.FullName);
// Output:
// System.Int32
```

Also, don't use the is operator to determine whether an instance is of a nullable value type. As the following example shows, you cannot distinguish types of a nullable value type instance and its underlying type instance with the `is` operator:

```
C#
```

```
int? a = 14;
if (a is int)
{
    Console.WriteLine("int? instance is compatible with int");
}

int b = 17;
if (b is int?)
{
    Console.WriteLine("int instance is compatible with int?");
}
// Output:
// int? instance is compatible with int
// int instance is compatible with int?
```

Instead use the Nullable.GetUnderlyingType from the first example and typeof operator to check if an instance is of a nullable value type.

> **Note**
>
> The methods described in this section are not applicable in the case of nullable reference types.

C# language specification

For more information, see the following sections of the C# language specification:

- Nullable types
- Lifted operators
- Implicit nullable conversions
- Explicit nullable conversions
- Lifted conversion operators

See also

- What exactly does 'lifted' mean?
- System.Nullable<T>

- System.Nullable
- Nullable.GetUnderlyingType
- Nullable reference types

Reference types (C# reference)

Article • 03/30/2024

There are two kinds of types in C#: reference types and value types. Variables of reference types store references to their data (objects), while variables of value types directly contain their data. With reference types, two variables can reference the same object; therefore, operations on one variable can affect the object referenced by the other variable. With value types, each variable has its own copy of the data, and it's not possible for operations on one variable to affect the other (except in the case of `in`, `ref`, and `out` parameter variables; see in, ref, and out parameter modifier).

The following keywords are used to declare reference types:

- class
- interface
- delegate
- record

C# also provides the following built-in reference types:

- dynamic
- object
- string

See also

- C# keywords
- Pointer types
- Value types

Built-in reference types (C# reference)

10/08/2025

C# has many built-in reference types. They have keywords or operators that are synonyms for a type in the .NET library.

The object type

The `object` type is an alias for System.Object in .NET. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from System.Object. You can assign values of any type (except `ref struct`, see ref struct) to variables of type `object`. Any `object` variable can be assigned to its default value using the literal `null`. When a variable of a value type is converted to object, it's *boxed*. When a variable of type `object` is converted to a value type, it's *unboxed*. For more information, see Boxing and Unboxing.

The string type

The `string` type represents a sequence of zero or more Unicode characters. `string` is an alias for System.String in .NET.

Although `string` is a reference type, the equality operators == and != are defined to compare the values of `string` objects, not references. Value based equality makes testing for string equality more intuitive. For example:

C#

```
string a = "hello";
string b = "h";
// Append to contents of 'b'
b += "ello";
Console.WriteLine(a == b);
Console.WriteLine(object.ReferenceEquals(a, b));
```

The previous example displays "True" and then "False" because the content of the strings is equivalent, but `a` and `b` don't refer to the same string instance.

The + operator concatenates strings:

C#

```
string a = "good " + "morning";
```

The preceding code creates a string object that contains "good morning".

Strings are *immutable*--the contents of a string object can't be changed after the object is created. For example, when you write this code, the compiler actually creates a new string object to hold the new sequence of characters, and that new object is assigned to `b`. The memory allocated for `b` (when it contained the string "h") is then eligible for garbage collection.

C#

```
string b = "h";
b += "ello";
```

The `[]` operator can be used for readonly access to individual characters of a string. Valid index values start at `0` and must be less than the length of the string:

C#

```
string str = "test";
char x = str[2];  // x = 's';
```

In similar fashion, the `[]` operator can also be used for iterating over each character in a string:

C#

```
string str = "test";

for (int i = 0; i < str.Length; i++)
{
  Console.Write(str[i] + " ");
}
// Output: t e s t
```

String literals

String literals are of type `string` and can be written in three forms, raw, quoted, and verbatim.

Raw string literals are available beginning in C# 11. Raw string literals can contain arbitrary text without requiring escape sequences. Raw string literals can include whitespace and new lines, embedded quotes, and other special characters. Raw string literals are enclosed in a minimum of three double quotation marks (""""):

C#

```
"""
This is a multi-line
    string literal with the second line indented.
"""
```

You can even include a sequence of three (or more) double quote characters. If your text requires an embedded sequence of quotes, you start and end the raw string literal with more quote marks, as needed:

C#

```
""""""
This raw string literal has four """", count them: """" four!
embedded quote characters in a sequence. That's why it starts and ends
with five double quotes.

You could extend this example with as many embedded quotes as needed for your
text.
""""""
```

Raw string literals typically have the starting and ending quote sequences on separate lines from the embedded text. Multiline raw string literals support strings that are themselves quoted strings:

C#

```
var message = """
"This is a very important message."
""";
Console.WriteLine(message);
// output: "This is a very important message."
```

When the starting and ending quotes are on separate lines, the newlines following the opening quote and preceding the ending quote aren't included in the final content. The closing quote sequence dictates the leftmost column for the string literal. You can indent a raw string literal to match the overall code format:

C#

```
var message = """
    "This is a very important message."
    """;
Console.WriteLine(message);
```

```
// output: "This is a very important message."
// The leftmost whitespace is not part of the raw string literal
```

Columns to the right of the ending quote sequence are preserved. This behavior enables raw strings for data formats such as JSON, YAML, or XML, as shown in the following example:

C#

```
var json= """
    {
        "prop": 0
    }
    """;
```

> **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.

The compiler issues an error if any of the text lines extend to the left of the closing quote sequence. The opening and closing quote sequences can be on the same line, providing the string literal neither starts nor ends with a quote character:

C#

```
var shortText = """He said "hello!" this morning.""";
```

You can combine raw string literals with string interpolation to include quote characters and braces in the output string.

Quoted string literals are enclosed in double quotation marks ("):

C#

```
"good morning"  // a string literal
```

String literals can contain any character literal. Escape sequences are included. The following example uses escape sequence `\\` for backslash, `\u0066` for the letter f, and `\n` for newline.

C#

```
string a = "\\\u0066\n F";
Console.WriteLine(a);
// Output:
// \f
//  F
```

> **Note**
>
> The escape code `\udddd` (where `dddd` is a four-digit number) represents the Unicode character U+`dddd`. Eight-digit Unicode escape codes are also recognized: `\Udddddddd`.

Verbatim string literals start with `@` and are also enclosed in double quotation marks. For example:

C#

```
@"good morning"  // a string literal
```

The advantage of verbatim strings is that escape sequences *aren't* processed, which makes it easy to write. For example, the following text matches a fully qualified Windows file name:

C#

```
@"c:\Docs\Source\a.txt"  // rather than "c:\\Docs\\Source\\a.txt"
```

To include a double quotation mark in an @-quoted string, double it:

C#

```
@"""Ahoy!"" cried the captain." // "Ahoy!" cried the captain.
```

UTF-8 string literals

Strings in .NET are stored using UTF-16 encoding. UTF-8 is the standard for Web protocols and other important libraries. Beginning in C# 11, you can add the `u8` suffix to a string literal to specify UTF-8 encoding. UTF-8 literals are stored as `ReadOnlySpan<byte>` objects. The natural type of a UTF-8 string literal is `ReadOnlySpan<byte>`. Using a UTF-8 string literal creates a more clear declaration than declaring the equivalent System.ReadOnlySpan<T>, as shown in the following code:

C#

```
ReadOnlySpan<byte> AuthWithTrailingSpace = new byte[] { 0x41, 0x55, 0x54, 0x48,
0x20 };
ReadOnlySpan<byte> AuthStringLiteral = "AUTH "u8;
```

To store a UTF-8 string literal as an array requires the use of ReadOnlySpan<T>.ToArray() to copy the bytes containing the literal to the mutable array:

C#

```
byte[] AuthStringLiteral = "AUTH "u8.ToArray();
```

UTF-8 string literals aren't compile time constants; they're runtime constants. Therefore, they can't be used as the default value for an optional parameter. UTF-8 string literals can't be combined with string interpolation. You can't use the `$` token and the `u8` suffix on the same string expression.

The delegate type

The declaration of a delegate type is similar to a method signature. It has a return value and any number of parameters of any type:

C#

```
public delegate void MessageDelegate(string message);
public delegate int AnotherDelegate(MyType m, long num);
```

In .NET, `System.Action` and `System.Func` types provide generic definitions for many common delegates. You likely don't need to define new custom delegate types. Instead, you can create

instantiations of the provided generic types.

A `delegate` is a built-in reference type that can be used to encapsulate a named or an anonymous method. Delegates are similar to function pointers in C++; however, delegates are type-safe and secure. For applications of delegates, see Delegates and Generic Delegates. Delegates are the basis for Events. A delegate can be instantiated by associating it either with a named or anonymous method.

The delegate must be instantiated with a method or lambda expression that has a compatible return type and input parameters. For more information on the degree of variance that is allowed in the method signature, see Variance in Delegates. For use with anonymous methods, the delegate and the code to be associated with it are declared together.

Delegate combination or removal fails with a runtime exception when the delegate types involved at run time are different due to variant conversion. The following example demonstrates a situation that fails:

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Valid due to implicit reference conversion of
// objectAction to Action<string>, but might fail
// at run time.
Action<string> combination = stringAction + objectAction;
```

You can create a delegate with the correct runtime type by creating a new delegate object. The following example demonstrates how this workaround might be applied to the preceding example.

C#

```
Action<string> stringAction = str => {};
Action<object> objectAction = obj => {};

// Creates a new delegate instance with a runtime type of Action<string>.
Action<string> wrappedObjectAction = new Action<string>(objectAction);

// The two Action<string> delegate instances can now be combined.
Action<string> combination = stringAction + wrappedObjectAction;
```

You can declare *function pointers*, which use similar syntax. A function pointer uses the `calli` instruction instead of instantiating a delegate type and calling the virtual `Invoke` method.

The dynamic type

The `dynamic` type indicates that use of the variable and references to its members bypass compile-time type checking. Instead, these operations are resolved at run time. The `dynamic` type simplifies access to COM APIs such as the Office Automation APIs, to dynamic APIs such as IronPython libraries, and to the HTML Document Object Model (DOM).

Type `dynamic` behaves like type `object` in most circumstances. In particular, any non-null expression can be converted to the `dynamic` type. The `dynamic` type differs from `object` in that operations that contain expressions of type `dynamic` aren't resolved or type checked by the compiler. The compiler packages together information about the operation, and that information is later used to evaluate the operation at run time. As part of the process, variables of type `dynamic` are compiled into variables of type `object`. Therefore, type `dynamic` exists only at compile time, not at run time.

The following example contrasts a variable of type `dynamic` to a variable of type `object`. To verify the type of each variable at compile time, place the mouse pointer over `dyn` or `obj` in the `WriteLine` statements. Copy the following code into an editor where IntelliSense is available. IntelliSense shows **dynamic** for `dyn` and **object** for `obj`.

C#

```
class Program
{
    static void Main(string[] args)
    {
        dynamic dyn = 1;
        object obj = 1;

        // Rest the mouse pointer over dyn and obj to see their
        // types at compile time.
        System.Console.WriteLine(dyn.GetType());
        System.Console.WriteLine(obj.GetType());
    }
}
```

The WriteLine statements display the run-time types of `dyn` and `obj`. At that point, both have the same type, integer. The following output is produced:

Console

```
System.Int32
System.Int32
```

To see the difference between `dyn` and `obj` at compile time, add the following two lines between the declarations and the `WriteLine` statements in the previous example.

C#

```
dyn = dyn + 3;
obj = obj + 3;
```

A compiler error is reported for the attempted addition of an integer and an object in expression `obj + 3`. However, no error is reported for `dyn + 3`. The expression that contains `dyn` isn't checked at compile time because the type of `dyn` is `dynamic`.

The following example uses `dynamic` in several declarations. The `Main` method also contrasts compile-time type checking with run-time type checking.

C#

```
using System;

namespace DynamicExamples
{
    class Program
    {
        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();
            Console.WriteLine(ec.ExampleMethod(10));
            Console.WriteLine(ec.ExampleMethod("value"));

            // The following line causes a compiler error because ExampleMethod
            // takes only one argument.
            //Console.WriteLine(ec.ExampleMethod(10, 4));

            dynamic dynamic_ec = new ExampleClass();
            Console.WriteLine(dynamic_ec.ExampleMethod(10));

            // Because dynamic_ec is dynamic, the following call to ExampleMethod
            // with two arguments does not produce an error at compile time.
            // However, it does cause a run-time error.
            //Console.WriteLine(dynamic_ec.ExampleMethod(10, 4));
        }
    }

    class ExampleClass
    {
        static dynamic _field;
        dynamic Prop { get; set; }

        public dynamic ExampleMethod(dynamic d)
        {
```

```
            dynamic local = "Local variable";
            int two = 2;

            if (d is int)
            {
                return local;
            }
            else
            {
                return two;
            }
        }
    }
}
// Results:
// Local variable
// 2
// Local variable
```

C# language specification

For more information, see the following sections of the C# language specification:

- §8.2.3 The object type
- §8.2.4 The dynamic type
- §8.2.5 The string type
- §8.2.8 Delegate types
- C# 11 - Raw string literals
- C# 11 - Raw string literals

See also

- C# Keywords
- Events
- Using Type dynamic
- Best Practices for Using Strings
- Basic String Operations
- Creating New Strings
- Type-testing and cast operators
- How to safely cast using pattern matching and the as and is operators
- Walkthrough: creating and using dynamic objects
- System.Object
- System.String
- System.Dynamic.DynamicObject

Records (C# reference)

08/19/2025

You use the `record` modifier to define a reference type that provides built-in functionality for encapsulating data. The `record class` syntax as a synonym to clarify a reference type, and `record struct` to define a value type with similar functionality.

When you declare a primary constructor on a record, the compiler generates public properties for the primary constructor parameters. The primary constructor parameters to a record are referred to as *positional parameters*. The compiler creates *positional properties* that mirror the primary constructor or positional parameters. The compiler doesn't synthesize properties for primary constructor parameters on types that don't have the `record` modifier.

The following two examples demonstrate `record` (or `record class`) reference types:

C#

```
public record Person(string FirstName, string LastName);
```

C#

```
public record Person
{
    public required string FirstName { get; init; }
    public required string LastName { get; init; }
};
```

The following two examples demonstrate `record struct` value types:

C#

```
public readonly record struct Point(double X, double Y, double Z);
```

C#

```
public record struct Point
{
    public double X { get; init; }
    public double Y { get; init; }
    public double Z { get; init; }
}
```

You can also create records with mutable properties and fields:

C#

```
public record Person
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
};
```

Record structs can be mutable as well, both positional record structs and record structs with no positional parameters:

C#

```
public record struct DataMeasurement(DateTime TakenAt, double Measurement);
```

C#

```
public record struct Point
{
    public double X { get; set; }
    public double Y { get; set; }
    public double Z { get; set; }
}
```

While records can be mutable, they're primarily intended for supporting immutable data models. The record type offers the following features:

- Concise syntax for creating a reference type with immutable properties
- Built-in behavior useful for a data-centric reference type:
  - Value equality
  - Concise syntax for nondestructive mutation
  - Built-in formatting for display
- Support for inheritance hierarchies

The preceding examples show some distinctions between records that are reference types and records that are value types:

- A `record` or a `record class` declares a reference type. The `class` keyword is optional, but can add clarity for readers. A `record struct` declares a value type.
- Positional properties are *immutable* in a `record class` and a `readonly record struct`. They're *mutable* in a `record struct`.

The remainder of this article discusses both `record class` and `record struct` types. The differences are detailed in each section. You should decide between a `record class` and a

`record struct` similar to deciding between a `class` and a `struct`. The term *record* is used to describe behavior that applies to all record types. Either `record struct` or `record class` is used to describe behavior that applies to only struct or class types, respectively.

Positional syntax for property and field definition

You can use positional parameters to declare properties of a record or to initialize property or field values. The following example creates a record with two positional properties:

C#

```
public record Person(string FirstName, string LastName);

public static void Main()
{
    Person person = new("Nancy", "Davolio");
    Console.WriteLine(person);
    // output: Person { FirstName = Nancy, LastName = Davolio }
}
```

When you use the positional syntax for property definition, the compiler creates:

- A public automatically implemented property for each positional parameter provided in the record declaration.
  - For `record` types and `readonly record struct` types: An init-only property.
  - For `record struct` types: A read-write property.
- A primary constructor whose parameters match the positional parameters on the record declaration.
- For record struct types, a parameterless constructor that sets each field to its default value.
- A `Deconstruct` method with an `out` parameter for each positional parameter provided in the record declaration. The method deconstructs properties defined by using positional syntax; it ignores properties that are defined by using standard property syntax.

You might want to add attributes to any of these elements the compiler creates from the record definition. You can add a *target* to any attribute you apply to the positional record's properties. The following example applies the System.Text.Json.Serialization.JsonPropertyNameAttribute to each property of the `Person` record. The `property:` target indicates that the attribute is applied to the compiler-generated property. Other values are `field:` to apply the attribute to the field, and `param:` to apply the attribute to the parameter.

C#

```
/// <summary>
/// Person record type
/// </summary>
/// <param name="FirstName">First Name</param>
/// <param name="LastName">Last Name</param>
/// <remarks>
/// The person type is a positional record containing the
/// properties for the first and last name. Those properties
/// map to the JSON elements "firstName" and "lastName" when
/// serialized or deserialized.
/// </remarks>
public record Person([property: JsonPropertyName("firstName")] string FirstName,
    [property: JsonPropertyName("lastName")] string LastName);
```

The preceding example also shows how to create XML documentation comments for the record. You can add the `<param>` tag to add documentation for the primary constructor's parameters.

If the generated automatically implemented property definition isn't what you want, you can define your own property or field of the same name. For example, you might want to change accessibility or mutability, or provide an implementation for either the `get` or `set` accessor. If you declare the member in your source, you must initialize it from the positional parameter of the record. If your property is an automatically implemented property, you must initialize the property. If you add a backing field in your source, you must initialize the backing field. The generated deconstructor uses your property or field definition. For instance, the following example declares the `FirstName` and `LastName` properties of a positional record `public`, but restricts the `Id` positional parameter to `internal`. You can use this syntax for records and record struct types.

C#

```
public record Person(string FirstName, string LastName, string Id)
{
    internal string Id { get; init; } = Id;
}

public static void Main()
{
    Person person = new("Nancy", "Davolio", "12345");
    Console.WriteLine(person.FirstName); //output: Nancy

}
```

If you want to create a field instead of a property, assign the positional parameter to a field, as shown in the following example:

```C#
public record Person(string FirstName, string LastName, string Id)
{
    internal readonly string Id = Id; // this.Id set to parameter Id
}

public static void Main()
{
    Person person = new("Nancy", "Davolio", "12345");
    Console.WriteLine(person.FirstName); //output: Nancy

}
```

A record type doesn't have to declare any positional properties. You can declare a record without any positional properties, and you can declare other fields and properties, as in the following example:

```C#
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; } = [];
};
```

Properties that the compiler generates from positional parameters are `public`. You declare the access modifiers on any properties you explicitly declare.

Immutability

A *positional record class* and a *positional readonly record struct* declare init-only properties. A *positional record struct* declares read-write properties. You can override either of those defaults, as shown in the previous section.

Immutability can be useful when you need a data-centric type to be thread-safe or you're depending on a hash code remaining the same in a hash table. Immutability isn't appropriate for all data scenarios, however. Entity Framework Core, for example, doesn't support updating with immutable entity types.

Init-only properties, whether created from positional parameters (`record class`, and `readonly record struct`) or by specifying `init` accessors, have *shallow immutability*. After initialization, you can't change the value of value-type properties or the reference of reference-type properties. However, the data that a reference-type property refers to can be changed. The following example shows that the content of a reference-type immutable property (an array in this case) is mutable:

```C#
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
{
    Person person = new("Nancy", "Davolio", new string[1] { "555-1234" });
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-1234

    person.PhoneNumbers[0] = "555-6789";
    Console.WriteLine(person.PhoneNumbers[0]); // output: 555-6789
}
```

The features unique to record types are implemented by compiler-synthesized methods, and none of these methods compromises immutability by modifying object state. Unless specified, the synthesized methods are generated for `record`, `record struct`, and `readonly record struct` declarations.

Value equality

If you don't override or replace equality methods, the type you declare governs how equality is defined:

- For `class` types, two objects are equal if they refer to the same object in memory.
- For `struct` types, two objects are equal if they are of the same type and store the same values.
- For types with the `record` modifier (`record class`, `record struct`, and `readonly record struct`), two objects are equal if they are of the same type and store the same values.

The definition of equality for a `record struct` is the same as for a `struct`. The difference is that for a `struct`, the implementation is in ValueType.Equals(Object) and relies on reflection. For records, the implementation is compiler synthesized and uses the declared data members.

Reference equality is required for some data models. For example, Entity Framework Core depends on reference equality to ensure that it uses only one instance of an entity type for what is conceptually one entity. For this reason, records and record structs aren't appropriate for use as entity types in Entity Framework Core.

The following example illustrates value equality of record types:

```C#
public record Person(string FirstName, string LastName, string[] PhoneNumbers);

public static void Main()
```

```
{
    var phoneNumbers = new string[2];
    Person person1 = new("Nancy", "Davolio", phoneNumbers);
    Person person2 = new("Nancy", "Davolio", phoneNumbers);
    Console.WriteLine(person1 == person2); // output: True

    person1.PhoneNumbers[0] = "555-1234";
    Console.WriteLine(person1 == person2); // output: True

    Console.WriteLine(ReferenceEquals(person1, person2)); // output: False
}
```

To implement value equality, the compiler synthesizes several methods, including:

- An override of Object.Equals(Object). It's an error if the override is declared explicitly.

  This method is used as the basis for the Object.Equals(Object, Object) static method when both parameters are non-null.

- A `virtual`, or `sealed`, `Equals(R? other)` where `R` is the record type. This method implements IEquatable<T>. This method can be declared explicitly.

- If the record type is derived from a base record type `Base`, `Equals(Base? other)`. It's an error if the override is declared explicitly. If you provide your own implementation of `Equals(R? other)`, provide an implementation of `GetHashCode` also.

- An override of Object.GetHashCode(). This method can be declared explicitly.

- Overrides of operators `==` and `!=`. It's an error if the operators are declared explicitly.

- If the record type is derived from a base record type, `protected override Type EqualityContract { get; };`. This property can be declared explicitly. For more information, see Equality in inheritance hierarchies.

The compiler doesn't synthesize a method when a record type has a method that matches the signature of a synthesized method allowed to be declared explicitly.

Nondestructive mutation

If you need to copy an instance with some modifications, you can use a `with` expression to achieve *nondestructive mutation*. A `with` expression makes a new record instance that is a copy of an existing record instance, with specified properties and fields modified. You use object initializer syntax to specify the values to be changed, as shown in the following example:

C#

```
public record Person(string FirstName, string LastName)
{
    public string[] PhoneNumbers { get; init; }
}

public static void Main()
{
    Person person1 = new("Nancy", "Davolio") { PhoneNumbers = new string[1] };
    Console.WriteLine(person1);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers =
System.String[] }

    Person person2 = person1 with { FirstName = "John" };
    Console.WriteLine(person2);
    // output: Person { FirstName = John, LastName = Davolio, PhoneNumbers =
System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { PhoneNumbers = new string[1] };
    Console.WriteLine(person2);
    // output: Person { FirstName = Nancy, LastName = Davolio, PhoneNumbers =
System.String[] }
    Console.WriteLine(person1 == person2); // output: False

    person2 = person1 with { };
    Console.WriteLine(person1 == person2); // output: True
}
```

The `with` expression can set positional properties or properties created by using standard property syntax. Explicitly declared properties must have an `init` or `set` accessor to be changed in a `with` expression.

The result of a `with` expression is a *shallow copy*, which means that for a reference property, only the reference to an instance is copied. Both the original record and the copy end up with a reference to the same instance.

To implement this feature for `record class` types, the compiler synthesizes a clone method and a copy constructor. The virtual clone method returns a new record initialized by the copy constructor. When you use a `with` expression, the compiler creates code that calls the clone method and then sets the properties that are specified in the `with` expression.

If you need different copying behavior, you can write your own copy constructor in a `record class`. If you do that, the compiler doesn't synthesize one. Make your constructor `private` if the record is `sealed`, otherwise make it `protected`. The compiler doesn't synthesize a copy constructor for `record struct` types. You can write one, but the compiler doesn't generate calls to it for `with` expressions. The values of the `record struct` are copied on assignment.

You can't override the clone method, and you can't create a member named `Clone` in any record type. The actual name of the clone method is compiler-generated.

> **ⓘ Important**
>
> In the preceding examples, all properties are independent. None of the properties are computed from other property values. A `with` expression first copies the existing record instance, then modifies any properties or fields specified in the `with` expression. Computed properties in `record` types should be computed on access, not initialized when the instance is created. Otherwise, a property could return the computed value based on the original instance, not the modified copy.

You ensure correctness on computed properties by computing the value on access, as shown in the following declaration:

C#

```
public record Point(int X, int Y)
{
    public double Distance => Math.Sqrt(X * X + Y * Y);
}
```

The preceding record type computes the `Distance` when accessed, as shown in the following example:

C#

```
Point p1 = new Point(3, 4);
Console.WriteLine($"Original point: {p1}");
p1 = p1 with { Y = 8 };
Console.WriteLine($"Modified point: {p1}");
// Output:
// Original point: Point { X = 3, Y = 4, Distance = 5 }
// Modified point: Point { X = 3, Y = 8, Distance = 8.54400374531753 }
```

Contrast that with the following declaration, where the `Distance` property is computed and cached as part of the initialization of a new instance:

C#

```
public record PointInit(int X, int Y)
{
    public double Distance { get; } = Math.Sqrt(X * X + Y * Y);
}
```

Because `Distance` is computed as part of initialization, the value is computed and cached before the `with` expression changes the value of `Y` in the copy. The result is that the distance is incorrect:

C#

```
PointInit pt1 = new PointInit(3, 4);
Console.WriteLine($"Original point: {pt1}");
pt1 = pt1 with { Y = 8 };
Console.WriteLine($"Incorrect Modified point: {pt1}");
// Output:
// Original point: PointInit { X = 3, Y = 4, Distance = 5 }
// Modified point: PointInit { X = 3, Y = 8, Distance = 5 }
```

The `Distance` computation isn't expensive to compute on each access. However, some computed properties might require access to more data or more extensive computation. In those cases, instead of a record, use a `class` type and compute the cached value when one of the components changes value.

Built-in formatting for display

Record types have a compiler-generated ToString method that displays the names and values of public properties and fields. The `ToString` method returns a string of the following format:

> <record type name> { <property name> = <value>, <property name> = <value>, ...}

The string printed for `<value>` is the string returned by the ToString() for the type of the property. In the following example, `ChildNames` is a System.Array, where `ToString` returns `System.String[]`:

```
Person { FirstName = Nancy, LastName = Davolio, ChildNames = System.String[] }
```

To implement this feature, in `record class` types, the compiler synthesizes a virtual `PrintMembers` method and a ToString override. In `record struct` types, this member is `private`. The `ToString` override creates a StringBuilder object with the type name followed by an opening bracket. It calls `PrintMembers` to add property names and values, then adds the closing bracket. The following example shows code similar to what the synthesized override contains:

C#

```
public override string ToString()
{
    StringBuilder stringBuilder = new StringBuilder();
    stringBuilder.Append("Teacher"); // type name
    stringBuilder.Append(" { ");
    if (PrintMembers(stringBuilder))
    {
        stringBuilder.Append(" ");
    }
    stringBuilder.Append("}");
    return stringBuilder.ToString();
}
```

You can provide your own implementation of `PrintMembers` or the `ToString` override. Examples are provided in the PrintMembers formatting in derived records section later in this article. Your implementation of `ToString` might include the `sealed` modifier, which prevents the compiler from synthesizing a `ToString` implementation for any derived records. You can create a consistent string representation throughout a hierarchy of `record` types. (Derived records still have a `PrintMembers` method generated for all derived properties.)

Inheritance

This section only applies to `record class` types.

A record can inherit from another record. However, a record can't inherit from a class, and a class can't inherit from a record.

Positional parameters in derived record types

The derived record declares positional parameters for all the parameters in the base record primary constructor. The base record declares and initializes those properties. The derived record doesn't hide them, but only creates and initializes properties for parameters that aren't declared in its base record.

The following example illustrates inheritance with positional property syntax:

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
```

```
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

Equality in inheritance hierarchies

This section applies to `record class` types, but not `record struct` types. For two record variables to be equal, the run-time type must be equal. The types of the containing variables might be different. Inherited equality comparison is illustrated in the following code example:

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Person student = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(teacher == student); // output: False

    Student student2 = new Student("Nancy", "Davolio", 3);
    Console.WriteLine(student2 == student); // output: True
}
```

In the example, all variables are declared as `Person`, even when the instance is a derived type of either `Student` or `Teacher`. The instances have the same properties and the same property values. But `student == teacher` returns `False` although both are `Person`-type variables, and `student == student2` returns `True` although one is a `Person` variable and one is a `Student` variable. The equality test depends on the runtime type of the actual object, not the declared type of the variable.

To implement this behavior, the compiler synthesizes an `EqualityContract` property that returns a Type object that matches the type of the record. The `EqualityContract` enables the equality methods to compare the runtime type of objects when they're checking for equality. If the base type of a record is `object`, this property is `virtual`. If the base type is another record type, this property is an override. If the record type is `sealed`, this property is effectively `sealed` because the type is `sealed`.

When code compares two instances of a derived type, the synthesized equality methods check all data members of the base and derived types for equality. The synthesized `GetHashCode` method uses the `GetHashCode` method from all data members declared in the base type and

the derived record type. The data members of a `record` include all declared fields and the compiler-synthesized backing field for any automatically implemented properties.

`with` expressions in derived records

The result of a `with` expression has the same run-time type as the expression's operand. All properties of the run-time type get copied, but you can only set properties of the compile-time type, as the following example shows:

C#

```
public record Point(int X, int Y)
{
    public int Zbase { get; set; }
};
public record NamedPoint(string Name, int X, int Y) : Point(X, Y)
{
    public int Zderived { get; set; }
};

public static void Main()
{
    Point p1 = new NamedPoint("A", 1, 2) { Zbase = 3, Zderived = 4 };

    Point p2 = p1 with { X = 5, Y = 6, Zbase = 7 }; // Can't set Name or Zderived
    Console.WriteLine(p2 is NamedPoint);  // output: True
    Console.WriteLine(p2);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = A, Zderived = 4 }

    Point p3 = (NamedPoint)p1 with { Name = "B", X = 5, Y = 6, Zbase = 7, Zderived
= 8 };
    Console.WriteLine(p3);
    // output: NamedPoint { X = 5, Y = 6, Zbase = 7, Name = B, Zderived = 8 }
}
```

`PrintMembers` formatting in derived records

The synthesized `PrintMembers` method of a derived record type calls the base implementation. The result is that all public properties and fields of both derived and base types are included in the `ToString` output, as shown in the following example:

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
```

```
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, Grade = 3 }
}
```

You can provide your own implementation of the `PrintMembers` method. If you do that, use the following signature:

- For a `sealed` record that derives from `object` (doesn't declare a base record): `private bool PrintMembers(StringBuilder builder)`;
- For a `sealed` record that derives from another record (note that the enclosing type is `sealed`, so the method is effectively `sealed`): `protected override bool PrintMembers(StringBuilder builder)`;
- For a record that isn't `sealed` and derives from object: `protected virtual bool PrintMembers(StringBuilder builder);`
- For a record that isn't `sealed` and derives from another record: `protected override bool PrintMembers(StringBuilder builder);`

Here's an example of code that replaces the synthesized `PrintMembers` methods, one for a record type that derives from object, and one for a record type that derives from another record:

C#

```
public abstract record Person(string FirstName, string LastName, string[]
PhoneNumbers)
{
    protected virtual bool PrintMembers(StringBuilder stringBuilder)
    {
        stringBuilder.Append($"FirstName = {FirstName}, LastName = {LastName}, ");
        stringBuilder.Append($"PhoneNumber1 = {PhoneNumbers[0]}, PhoneNumber2 =
{PhoneNumbers[1]}");
        return true;
    }
}

public record Teacher(string FirstName, string LastName, string[] PhoneNumbers,
int Grade)
    : Person(FirstName, LastName, PhoneNumbers)
{
    protected override bool PrintMembers(StringBuilder stringBuilder)
    {
        if (base.PrintMembers(stringBuilder))
        {
```

```
            stringBuilder.Append(", ");
        }
        ;
        stringBuilder.Append($"Grade = {Grade}");
        return true;
    }
};

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", new string[2] { "555-1234",
"555-6789" }, 3);
    Console.WriteLine(teacher);
    // output: Teacher { FirstName = Nancy, LastName = Davolio, PhoneNumber1 =
555-1234, PhoneNumber2 = 555-6789, Grade = 3 }
}
```

> **Note**
>
> The compiler synthesizes `PrintMembers` in derived records even when a base record sealed the `ToString` method. You can also create your own implementation of `PrintMembers`.

Deconstructor behavior in derived records

The `Deconstruct` method of a derived record returns the values of all positional properties of the compile-time type. If the variable type is a base record, only the base record properties are deconstructed unless the object is cast to the derived type. The following example demonstrates calling a deconstructor on a derived record.

C#

```
public abstract record Person(string FirstName, string LastName);
public record Teacher(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);
public record Student(string FirstName, string LastName, int Grade)
    : Person(FirstName, LastName);

public static void Main()
{
    Person teacher = new Teacher("Nancy", "Davolio", 3);
    var (firstName, lastName) = teacher; // Doesn't deconstruct Grade
    Console.WriteLine($"{firstName}, {lastName}");// output: Nancy, Davolio

    var (fName, lName, grade) = (Teacher)teacher;
    Console.WriteLine($"{fName}, {lName}, {grade}");// output: Nancy, Davolio, 3
}
```

Generic constraints

The `record` keyword is a modifier for either a `class` or `struct` type. Adding the `record` modifier includes the behavior described earlier in this article. There's no generic constraint that requires a type to be a record. A `record class` satisfies the `class` constraint. A `record struct` satisfies the `struct` constraint. For more information, see Constraints on type parameters.

C# language specification

For more information, see the Classes section of the C# language specification.

For more information about these features, see the following feature proposal notes:

- Records
- Init-only setters
- Covariant returns

See also

- Design guidelines - Choosing between class and struct
- Design guidelines - Struct design
- The C# type system
- with expression

class (C# Reference)

Article • 03/30/2024

Classes are declared using the keyword `class`, as shown in the following example:

```
C#
```

```
class TestClass
{
    // Methods, properties, fields, events, delegates
    // and nested classes go here.
}
```

Remarks

Only single inheritance is allowed in C#. In other words, a class can inherit implementation from one base class only. However, a class can implement more than one interface. The following table shows examples of class inheritance and interface implementation:

Expand table

Inheritance	Example
None	`class ClassA { }`
Single	`class DerivedClass : BaseClass { }`
None, implements two interfaces	`class ImplClass : IFace1, IFace2 { }`
Single, implements one interface	`class ImplDerivedClass : BaseClass, IFace1 { }`

Classes that you declare directly within a namespace, not nested within other classes, can be either public or internal. Classes are `internal` by default.

Class members, including nested classes, can be public, protected internal, protected, internal, private, or private protected. Members are `private` by default.

For more information, see Access Modifiers.

You can declare generic classes that have type parameters. For more information, see Generic Classes.

A class can contain declarations of the following members:

- Constructors
- Constants
- Fields
- Finalizers
- Methods
- Properties
- Indexers
- Operators
- Events
- Delegates
- Classes
- Interfaces
- Structure types
- Enumeration types

Example

The following example demonstrates declaring class fields, constructors, and methods. It also demonstrates object instantiation and printing instance data. In this example, two classes are declared. The first class, `Child`, contains two private fields (`name` and `age`), two public constructors and one public method. The second class, `StringTest`, is used to contain `Main`.

C#

```
class Child
{
    private int age;
    private string name;

    // Default constructor:
    public Child()
    {
        name = "N/A";
    }
```

```
    // Constructor:
    public Child(string name, int age)
    {
        this.name = name;
        this.age = age;
    }

    // Printing method:
    public void PrintChild()
    {
        Console.WriteLine("{0}, {1} years old.", name, age);
    }
}

class StringTest
{
    static void Main()
    {
        // Create objects by using the new operator:
        Child child1 = new Child("Craig", 11);
        Child child2 = new Child("Sally", 10);

        // Create an object using the default constructor:
        Child child3 = new Child();

        // Display results:
        Console.Write("Child #1: ");
        child1.PrintChild();
        Console.Write("Child #2: ");
        child2.PrintChild();
        Console.Write("Child #3: ");
        child3.PrintChild();
    }
}
/* Output:
    Child #1: Craig, 11 years old.
    Child #2: Sally, 10 years old.
    Child #3: N/A, 0 years old.
*/
```

Comments

Notice that in the previous example the private fields (`name` and `age`) can only be accessed through the public method of the `Child` class. For example, you cannot print the child's name, from the `Main` method, using a statement like this:

C#

```
Console.Write(child1.name);   // Error
```

Accessing private members of `Child` from `Main` would only be possible if `Main` were a member of the class.

Types declared inside a class without an access modifier default to `private`, so the data members in this example would still be `private` if the keyword were removed.

Finally, notice that for the object created using the parameterless constructor (`child3`), the `age` field was initialized to zero by default.

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

- C# Keywords
- Reference Types

interface (C# Reference)

10/04/2025

An interface defines a contract. Any class, record or struct that implements that contract must provide an implementation of the members defined in the interface.

An interface can define a default implementation for a member. It can also define static members to provide a single implementation for common functionality.

Beginning with C# 11, an interface can define `static abstract` or `static virtual` members to declare that an implementing type must provide the declared members. Typically, `static virtual` methods declare that an implementation must define a set of overloaded operators.

In the following example, class `ImplementationClass` must implement a method named `SampleMethod` that has no parameters and returns `void`.

C#

```
interface ISampleInterface
{
    void SampleMethod();
}

class ImplementationClass : ISampleInterface
{
    // Explicit interface member implementation:
    void ISampleInterface.SampleMethod()
    {
        // Method implementation.
    }

    static void Main()
    {
        // Declare an interface instance.
        ISampleInterface obj = new ImplementationClass();

        // Call the member.
        obj.SampleMethod();
    }
}
```

For more information and examples, see Interfaces.

Access modifiers

An interface can be a member of a namespace or a class. A top-level interface, one declared in a namespace but not nested inside another type, can be declared `public` or `internal`. The default is `internal`. Nested interface declarations, those declared inside another type, can be declared using any access modifier.

Interface members *without* an implementation (abstract members) are implicitly `public` and cannot have any other access modifier. Interface members *with* a default implementation are `private` by default if no access modifier is specified, but can be declared with any access modifier (`public`, `private`, `protected`, or `internal`).

Interface members

An interface declaration can contain the following members:

- Methods.
- Properties.
- Indexers.
- Events.
- Constants.
- Operators.
- A static constructor.
- Nested types.
- Static fields, methods, properties, indexers, and events.
- Member declarations using the explicit interface implementation syntax.
- Explicit access modifiers (the default access for abstract methods is public).

Default interface members

Member declarations typically don't contain a body, however, an interface member can declare a body. Member bodies in an interface are the *default implementation*. Members with bodies permit the interface to provide a "default" implementation for classes and structs that don't provide an overriding implementation.

> **Important**
>
> Adding default interfaces members forces any `ref struct` that implements the interface to add an explicit declaration of that member.

Static abstract and virtual members

Beginning with C# 11, an interface can declare `static abstract` and `static virtual` members for all member types except fields. Interfaces can declare that implementing types must define operators or other static members. This feature enables generic algorithms to specify number-like behavior. You can see examples in the numeric types in the .NET runtime, such as System.Numerics.INumber<TSelf>. These interfaces define common mathematical operators that are implemented by many numeric types. The compiler must resolve calls to `static virtual` and `static abstract` methods at compile time. The `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. Therefore, `static virtual` methods are almost exclusively 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. For example, the `INumber<T>` interface declares that `T` must implement `INumber<T>`. The compiler uses the type argument to resolve calls to the methods and operators declared in the interface declaration. For example, the `int` type implements `INumber<int>`. When the type parameter `T` denotes the type argument `int`, the `static` members declared on `int` are invoked. Alternatively, when `double` is the type argument, the `static` members declared on the `double` type are invoked.

> **Important**
>
> Method dispatch for `static abstract` and `static virtual` methods declared in interfaces is resolved using the compile time type of an expression. If the runtime type of an expression is derived from a different compile time type, the static methods on the base (compile time) type will be called.

You can try this feature by working with the tutorial on static abstract members in interfaces.

Interface inheritance

Interfaces can't contain instance state. While static fields are now permitted, instance fields aren't permitted in interfaces. Instance auto-properties aren't supported in interfaces, as they would implicitly declare a hidden field. This rule has a subtle effect on property declarations. In an interface declaration, the following code doesn't declare an automatically implemented property as it does in a `class` or `struct`. Instead, it declares a property that doesn't have a default implementation but must be implemented in any type that implements the interface:

```
C#
```

```
public interface INamed
{
  public string Name {get; set;}
}
```

An interface can inherit from one or more base interfaces. When an interface inherits from another interface, a type implementing the derived interface must implement all the members in the base interfaces as well as those declared in the derived interface, as shown in the following code:

C#

```
public interface I1
{
    void M1();
}

public interface I2 : I1
{
    void M2();
}

public class C : I2
{
    // implements I1.M1
    public void M1() { }
    // implements I2.M2
    public void M2() { }
}
```

When an interface overrides a method implemented in a base interface, it must use the explicit interface implementation syntax.

When a base type list contains a base class and interfaces, the base class must come first in the list.

A class that implements an interface can explicitly implement members of that interface. An explicitly implemented member can't be accessed through a class instance, but only through an instance of the interface. In addition, default interface members can only be accessed through an instance of the interface.

For more information about explicit interface implementation, see Explicit Interface Implementation.

Example interface implementation

The following example demonstrates interface implementation. In this example, the interface contains the property declaration and the class contains the implementation. Any instance of a class that implements `IPoint` has integer properties `x` and `y`.

C#

```
interface IPoint
{
    // Property signatures:
    int X { get; set; }

    int Y { get; set; }

    double Distance { get; }
}

class Point : IPoint
{
    // Constructor:
    public Point(int x, int y)
    {
        X = x;
        Y = y;
    }

    // Property implementation:
    public int X { get; set; }

    public int Y { get; set; }

    // Property implementation
    public double Distance =>
       Math.Sqrt(X * X + Y * Y);
}

class MainClass
{
    static void PrintPoint(IPoint p)
    {
        Console.WriteLine("x={0}, y={1}", p.X, p.Y);
    }

    static void Main()
    {
        IPoint p = new Point(2, 3);
        Console.Write("My Point: ");
        PrintPoint(p);
    }
}
// Output: My Point: x=2, y=3
```

C# language specification

For more information, see the Interfaces section of the C# language specification, the feature specification for C# 8 - Default interface members, and the feature spec for C# 11 - static abstract members in interfaces.

See also

- C# Keywords
- Reference Types
- Interfaces
- Using Properties
- Using Indexers

Nullable reference types (C# reference)

> **Note**
>
> This article covers nullable reference types. You can also declare nullable value types.

Nullable reference types are available in code that's in a *nullable aware context*. Nullable reference types, the null static analysis warnings, and the null-forgiving operator are optional language features. All are turned off by default. A *nullable context* is controlled at the project level using build settings, or in code using pragmas.

> **Important**
>
> All project templates enable the *nullable context* for the project. Projects created with earlier templates don't include this element, and these features are off unless you enable them in the project file or use pragmas.

In a nullable aware context:

- A variable of a reference type `T` must be initialized with non-null, and can never be assigned a value that might be `null`.
- A variable of a reference type `T?` can be initialized with `null` or assigned `null`, but is required to be checked against `null` before dereferencing.
- A variable `m` of type `T?` is considered to be non-null when you apply the null-forgiving operator, as in `m!`.

The compiler enforces the distinctions between a non-nullable reference type `T` and a nullable reference type `T?` using the preceding rules. A variable of type `T` and a variable of type `T?` are the same .NET type. The following example declares a non-nullable string and a nullable string, and then uses the null-forgiving operator to assign a value to a non-nullable string:

C#

```
string notNull = "Hello";
string? nullable = default;
notNull = nullable!; // null forgiveness
```

The variables `notNull` and `nullable` are both represented by the String type. Because the non-nullable and nullable types are both stored as the same type, there are several locations where using a nullable reference type isn't allowed. In general, a nullable reference type can't be used as a base class or implemented interface. A nullable reference type can't be used in any object

creation or type testing expression. A nullable reference type can't be the type of a member access expression. The following examples show these constructs:

C#

```
public MyClass : System.Object? // not allowed
{
}

var nullEmpty = System.String?.Empty; // Not allowed
var maybeObject = new object?(); // Not allowed
try
{
    if (thing is string? nullableString) // not allowed
        Console.WriteLine(nullableString);
} catch (Exception? e) // Not Allowed
{
    Console.WriteLine("error");
}
```

Nullable references and static analysis

The examples in the previous section illustrate the nature of nullable reference types. Nullable reference types aren't new class types, but rather annotations on existing reference types. The compiler uses those annotations to help you find potential null reference errors in your code. There's no runtime difference between a non-nullable reference type and a nullable reference type. The compiler doesn't add any runtime checking for non-nullable reference types. The benefits are in the compile-time analysis. The compiler generates warnings that help you find and fix potential null errors in your code. You declare your intent, and the compiler warns you when your code violates that intent.

> **ⓘ Important**
>
> Nullable reference annotations don't introduce behavior changes, but other libraries may use reflection to produce different runtime behavior for nullable and non-nullable reference types. Notably, Entity Framework Core reads nullable attributes. It interprets a nullable reference as an optional value, and a non-nullable reference as a required value.

In a nullable enabled context, the compiler performs static analysis on variables of any reference type, both nullable and non-nullable. The compiler tracks the *null-state* of each reference variable as either *not-null* or *maybe-null*. The default state of a non-nullable reference is *not-null*. The default state of a nullable reference is *maybe-null*.

Non-nullable reference types should always be safe to dereference because their *null-state* is *not-null*. To enforce that rule, the compiler issues warnings if a non-nullable reference type isn't initialized to a non-null value. Local variables must be assigned where they're declared. Every field must be assigned a *not-null* value, in a field initializer or every constructor. The compiler issues warnings when a non-nullable reference is assigned to a reference whose state is *maybe-null*. Generally, a non-nullable reference is *not-null* and no warnings are issued when those variables are dereferenced.

> **Note**
>
> If you assign a *maybe-null* expression to a non-nullable reference type, the compiler generates a warning. The compiler then generates warnings for that variable until it's assigned to a *not-null* expression.

Nullable reference types can be initialized or assigned to `null`. Therefore, static analysis must determine that a variable is *not-null* before it's dereferenced. If a nullable reference is determined to be *maybe-null*, assigning to a non-nullable reference variable generates a compiler warning. The following class shows examples of these warnings:

C#

```
public class ProductDescription
{
    private string shortDescription;
    private string? detailedDescription;

    public ProductDescription() // Warning! shortDescription not initialized.
    {
    }

    public ProductDescription(string productDescription) =>
        this.shortDescription = productDescription;

    public void SetDescriptions(string productDescription, string? details=null)
    {
        shortDescription = productDescription;
        detailedDescription = details;
    }

    public string GetDescription()
    {
        if (detailedDescription.Length == 0) // Warning! dereference possible null
        {
            return shortDescription;
        }
        else
        {
            return $"{shortDescription}\n{detailedDescription}";
```

```
        }
    }

    public string FullDescription()
    {
        if (detailedDescription == null)
        {
            return shortDescription;
        }
        else if (detailedDescription.Length > 0) // OK, detailedDescription can't be
null.
        {
            return $"{shortDescription}\n{detailedDescription}";
        }
        return shortDescription;
    }
}
```

The following snippet shows where the compiler emits warnings when using this class:

C#

```
string shortDescription = default; // Warning! non-nullable set to null;
var product = new ProductDescription(shortDescription); // Warning! static analysis
knows shortDescription maybe null.

string description = "widget";
var item = new ProductDescription(description);

item.SetDescriptions(description, "These widgets will do everything.");
```

The preceding examples demonstrate how compiler's static analysis determines the *null-state* of reference variables. The compiler applies language rules for null checks and assignments to inform its analysis. The compiler can't make assumptions about the semantics of methods or properties. If you call methods that perform null checks, the compiler can't know those methods affect a variable's *null-state*. There are attributes you can add to your APIs to inform the compiler about the semantics of arguments and return values. Many common APIs in the .NET libraries have these attributes. For example, the compiler correctly interprets IsNullOrEmpty as a null check. For more information about the attributes that apply to *null-state* static analysis, see the article on Nullable attributes.

Setting the nullable context

There are two ways to control the nullable context. At the project level, you can add the `<Nullable>enable</Nullable>` project setting. In a single C# source file, you can add the `#nullable enable` pragma to enable the nullable context. See the article on setting a nullable

strategy. Before .NET 6, new projects use the default, `<Nullable>disable</Nullable>`. Beginning with .NET 6, new projects include the `<Nullable>enable</Nullable>` element in the project file.

C# language specification

For more information, see the Nullable reference types section of the C# language specification:

See also

- Nullable value types

Last updated on 11/01/2025

Collections

Article • 02/04/2025

The .NET runtime provides many collection types that store and manage groups of related objects. Some of the collection types, such as System.Array, System.Span<T>, and System.Memory<T> are recognized in the C# language. In addition, interfaces like System.Collections.Generic.IEnumerable<T> are recognized in the language for enumerating the elements of a collection.

Collections provide a flexible way to work with groups of objects. You can classify different collections by these characteristics:

- **Element access**: Every collection can be enumerated to access each element in order. Some collections access elements by *index*, the element's position in an ordered collection. The most common example is System.Collections.Generic.List<T>. Other collections access elements by *key*, where a *value* is associated with a single *key*. The most common example is System.Collections.Generic.Dictionary<TKey,TValue>. You choose between these collection types based on how your app accesses elements.
- **Performance profile**: Every collection has different performance profiles for actions like adding an element, finding an element, or removing an element. You can pick a collection type based on the operations used most in your app.
- **Grow and shrink dynamically**: Most collections support adding or removing elements dynamically. Notably, Array, System.Span<T>, and System.Memory<T> don't.

In addition to those characteristics, the runtime provides specialized collections that prevent adding or removing elements or modifying the elements of the collection. Other specialized collections provide safety for concurrent access in multi-threaded apps.

You can find all the collection types in the .NET API reference. For more information, see Commonly Used Collection Types and Selecting a Collection Class.

> **Note**
>
> For the examples in this article, you might need to add **using directives** for the `System.Collections.Generic` and `System.Linq` namespaces.

Arrays are represented by System.Array and have syntax support in the C# language. This syntax provides more concise declarations for array variables.

System.Span<T> is a ref struct type that provides a snapshot over a sequence of elements without copying those elements. The compiler enforces safety rules to ensure the `Span` can't be accessed after the sequence it references is no longer in scope. It's used in many .NET APIs to improve performance. Memory<T> provides similar behavior when you can't use a `ref struct` type.

Beginning with C# 12, all of the collection types can be initialized using a Collection expression.

Indexable collections

An *indexable collection* is one where you can access each element using its index. Its *index* is the number of elements before it in the sequence. Therefore, the element reference by index `0` is the first element, index `1` is the second, and so on. These examples use the List<T> class. It's the most common indexable collection.

The following example creates and initializes a list of strings, removes an element, and adds an element to the end of the list. After each modification, it iterates through the strings by using a foreach statement or a `for` loop:

C#

```
// Create a list of strings by using a
// collection initializer.
List<string> salmons = ["chinook", "coho", "pink", "sockeye"];

// Iterate through the list.
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook coho pink sockeye

// Remove an element from the list by specifying
// the object.
salmons.Remove("coho");


// Iterate using the index:
for (var index = 0; index < salmons.Count; index++)
{
    Console.Write(salmons[index] + " ");
}
// Output: chinook pink sockeye

// Add the removed element
salmons.Add("coho");
// Iterate through the list.
```

```
foreach (var salmon in salmons)
{
    Console.Write(salmon + " ");
}
// Output: chinook pink sockeye coho
```

The following example removes elements from a list by index. Instead of a `foreach` statement, it uses a `for` statement that iterates in descending order. The RemoveAt method causes elements after a removed element to have a lower index value.

C#

```
List<int> numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];

// Remove odd numbers.
for (var index = numbers.Count - 1; index >= 0; index--)
{
    if (numbers[index] % 2 == 1)
    {
        // Remove the element by specifying
        // the zero-based index in the list.
        numbers.RemoveAt(index);
    }
}

// Iterate through the list.
// A lambda expression is placed in the ForEach method
// of the List(T) object.
numbers.ForEach(
    number => Console.Write(number + " "));
// Output: 0 2 4 6 8
```

For the type of elements in the List<T>, you can also define your own class. In the following example, the `Galaxy` class that is used by the List<T> is defined in the code.

C#

```
private static void IterateThroughList()
{
    var theGalaxies = new List<Galaxy>
    {
        new (){ Name="Tadpole", MegaLightYears=400},
        new (){ Name="Pinwheel", MegaLightYears=25},
        new (){ Name="Milky Way", MegaLightYears=0},
        new (){ Name="Andromeda", MegaLightYears=3}
    };

    foreach (Galaxy theGalaxy in theGalaxies)
    {
        Console.WriteLine(theGalaxy.Name + "  " + theGalaxy.MegaLightYears);
```

```
    }

    // Output:
    //  Tadpole  400
    //  Pinwheel  25
    //  Milky Way  0
    //  Andromeda  3
}

public class Galaxy
{
    public string Name { get; set; }
    public int MegaLightYears { get; set; }
}
```

Key/value pair collections

These examples use the Dictionary<TKey,TValue> class. It's the most common dictionary collection. A dictionary collection enables you to access elements in the collection by using the key of each element. Each addition to the dictionary consists of a value and its associated key.

The following example creates a `Dictionary` collection and iterates through the dictionary by using a `foreach` statement.

C#

```
private static void IterateThruDictionary()
{
    Dictionary<string, Element> elements = BuildDictionary();

    foreach (KeyValuePair<string, Element> kvp in elements)
    {
        Element theElement = kvp.Value;

        Console.WriteLine("key: " + kvp.Key);
        Console.WriteLine("values: " + theElement.Symbol + " " +
            theElement.Name + " " + theElement.AtomicNumber);
    }
}

public class Element
{
    public required string Symbol { get; init; }
    public required string Name { get; init; }
    public required int AtomicNumber { get; init; }
}

private static Dictionary<string, Element> BuildDictionary() =>
    new ()
```

```
    {
        {"K",
            new (){ Symbol="K", Name="Potassium", AtomicNumber=19}},
        {"Ca",
            new (){ Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        {"Sc",
            new (){ Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        {"Ti",
            new (){ Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
```

The following example uses the ContainsKey method and the Item[] property of `Dictionary` to quickly find an item by key. The `Item` property enables you to access an item in the `elements` collection by using the `elements[symbol]` in C#.

C#

```
if (elements.ContainsKey(symbol) == false)
{
    Console.WriteLine(symbol + " not found");
}
else
{
    Element theElement = elements[symbol];
    Console.WriteLine("found: " + theElement.Name);
}
```

The following example instead uses the TryGetValue method to quickly find an item by key.

C#

```
if (elements.TryGetValue(symbol, out Element? theElement) == false)
    Console.WriteLine(symbol + " not found");
else
    Console.WriteLine("found: " + theElement.Name);
```

Iterators

An *iterator* is used to perform a custom iteration over a collection. An iterator can be a method or a `get` accessor. An iterator uses a yield return statement to return each element of the collection one at a time.

You call an iterator by using a foreach statement. Each iteration of the `foreach` loop calls the iterator. When a `yield return` statement is reached in the iterator, an expression is

returned, and the current location in code is retained. Execution is restarted from that location the next time that the iterator is called.

For more information, see Iterators (C#).

The following example uses an iterator method. The iterator method has a `yield return` statement that is inside a `for` loop. In the `ListEvenNumbers` method, each iteration of the `foreach` statement body creates a call to the iterator method, which proceeds to the next `yield return` statement.

C#

```
private static void ListEvenNumbers()
{
    foreach (int number in EvenSequence(5, 18))
    {
        Console.Write(number.ToString() + " ");
    }
    Console.WriteLine();
    // Output: 6 8 10 12 14 16 18
}

private static IEnumerable<int> EvenSequence(
    int firstNumber, int lastNumber)
{
    // Yield even numbers in the range.
    for (var number = firstNumber; number <= lastNumber; number++)
    {
        if (number % 2 == 0)
        {
            yield return number;
        }
    }
}
```

LINQ and collections

Language-integrated query (LINQ) can be used to access collections. LINQ queries provide filtering, ordering, and grouping capabilities. For more information, see Getting Started with LINQ in C#.

The following example runs a LINQ query against a generic `List`. The LINQ query returns a different collection that contains the results.

C#

```
private static void ShowLINQ()
{
    List<Element> elements = BuildList();

    // LINQ Query.
    var subset = from theElement in elements
                 where theElement.AtomicNumber < 22
                 orderby theElement.Name
                 select theElement;

    foreach (Element theElement in subset)
    {
        Console.WriteLine(theElement.Name + " " + theElement.AtomicNumber);
    }

    // Output:
    //  Calcium 20
    //  Potassium 19
    //  Scandium 21
}

private static List<Element> BuildList() => new()
    {
        { new(){ Symbol="K", Name="Potassium", AtomicNumber=19}},
        { new(){ Symbol="Ca", Name="Calcium", AtomicNumber=20}},
        { new(){ Symbol="Sc", Name="Scandium", AtomicNumber=21}},
        { new(){ Symbol="Ti", Name="Titanium", AtomicNumber=22}}
    };
```

Arrays

Article • 12/14/2024

You can store multiple variables of the same type in an array data structure. You declare an array by specifying the type of its elements. If you want the array to store elements of any type, you can specify `object` as its type. In the unified type system of C#, all types, predefined and user-defined, reference types and value types, inherit directly or indirectly from Object.

C#

```
type[] arrayName;
```

An array is a reference type, so the array can be a nullable reference type. The element types might be reference types, so an array can be declared to hold nullable reference types. The following example declarations show the different syntax used to declare the nullability of the array or the elements:

C#

```
type?[] arrayName; // non nullable array of nullable element types.
type[]? arrayName; // nullable array of non-nullable element types.
type?[]? arrayName; // nullable array of nullable element types.
```

Uninitialized elements in an array are set to the default value for that type:

C#

```
int[] numbers = new int[10]; // All values are 0
string[] messages = new string[10]; // All values are null.
```

> **Important**
>
> In the preceding example, even though the type is `string[]`, an array of non-nullable strings, the default value for each element is null. The best way to initialize an array to non-null values is to use a collection expressions.

An array has the following properties:

- An array can be single-dimensional, multidimensional, or jagged.

- The number of dimensions are set when an array variable is declared. The length of each dimension is established when the array instance is created. These values can't be changed during the lifetime of the instance.
- A jagged array is an array of arrays, and each member array has the default value of `null`.
- Arrays are zero indexed: an array with `n` elements is indexed from `0` to `n-1`.
- Array elements can be of any type, including an array type.
- Array types are reference types derived from the abstract base type Array. All arrays implement IList and IEnumerable. You can use the foreach statement to iterate through an array. Single-dimensional arrays also implement IList<T> and IEnumerable<T>.

The elements of an array can be initialized to known values when the array is created. Beginning with C# 12, all of the collection types can be initialized using a Collection expression. Elements that aren't initialized are set to the default value. The default value is the 0-bit pattern. All reference types (including non-nullable types), have the values `null`. All value types have the 0-bit patterns. That means the Nullable<T>.HasValue property is `false` and the Nullable<T>.Value property is undefined. In the .NET implementation, the `Value` property throws an exception.

The following example creates single-dimensional, multidimensional, and jagged arrays:

C#

```
// Declare a single-dimensional array of 5 integers.
int[] array1 = new int[5];

// Declare and set array element values.
int[] array2 = [1, 2, 3, 4, 5, 6];

// Declare a two dimensional array.
int[,] multiDimensionalArray1 = new int[2, 3];

// Declare and set array element values.
int[,] multiDimensionalArray2 = { { 1, 2, 3 }, { 4, 5, 6 } };

// Declare a jagged array.
int[][] jaggedArray = new int[6][];

// Set the values of the first array in the jagged array structure.
jaggedArray[0] = [1, 2, 3, 4];
```

> **Important**

Many of the examples in this article use collection expressions (which use square brackets) to initialize the arrays. Collection expressions were first introduced in C# 12, which shipped with .NET 8. If you can't upgrade to C# 12 yet, use `{` and `}` to initialize the arrays instead.

C#

```
// Collection expressions:
int[] array = [1, 2, 3, 4, 5, 6];
// Alternative syntax:
int[] array2 = {1, 2, 3, 4, 5, 6};
```

Single-dimensional arrays

A *single-dimensional array* is a sequence of like elements. You access an element via its *index*. The *index* is its ordinal position in the sequence. The first element in the array is at index `0`. You create a single-dimensional array using the new operator specifying the array element type and the number of elements. The following example declares and initializes single-dimensional arrays:

C#

```
int[] array = new int[5];
string[] weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];

Console.WriteLine(weekDays[0]);
Console.WriteLine(weekDays[1]);
Console.WriteLine(weekDays[2]);
Console.WriteLine(weekDays[3]);
Console.WriteLine(weekDays[4]);
Console.WriteLine(weekDays[5]);
Console.WriteLine(weekDays[6]);

/*Output:
Sun
Mon
Tue
Wed
Thu
Fri
Sat
*/
```

The first declaration declares an uninitialized array of five integers, from `array[0]` to `array[4]`. The elements of the array are initialized to the default value of the element

type, `0` for integers. The second declaration declares an array of strings and initializes all seven values of that array. A series of `Console.WriteLine` statements prints all the elements of the `weekDay` array. For single-dimensional arrays, the `foreach` statement processes elements in increasing index order, starting with index 0 and ending with index `Length - 1`.

Pass single-dimensional arrays as arguments

You can pass an initialized single-dimensional array to a method. In the following example, an array of strings is initialized and passed as an argument to a `DisplayArray` method for strings. The method displays the elements of the array. Next, the `ChangeArray` method reverses the array elements, and then the `ChangeArrayElements` method modifies the first three elements of the array. After each method returns, the `DisplayArray` method shows that passing an array by value doesn't prevent changes to the array elements.

C#

```
class ArrayExample
{
    static void DisplayArray(string[] arr) =>
Console.WriteLine(string.Join(" ", arr));

    // Change the array by reversing its elements.
    static void ChangeArray(string[] arr) => Array.Reverse(arr);

    static void ChangeArrayElements(string[] arr)
    {
        // Change the value of the first three array elements.
        arr[0] = "Mon";
        arr[1] = "Wed";
        arr[2] = "Fri";
    }

    static void Main()
    {
        // Declare and initialize an array.
        string[] weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri",
"Sat"];
        // Display the array elements.
        DisplayArray(weekDays);
        Console.WriteLine();

        // Reverse the array.
        ChangeArray(weekDays);
        // Display the array again to verify that it stays reversed.
        Console.WriteLine("Array weekDays after the call to ChangeArray:");
        DisplayArray(weekDays);
```

```
        Console.WriteLine();

        // Assign new values to individual array elements.
        ChangeArrayElements(weekDays);
        // Display the array again to verify that it has changed.
        Console.WriteLine("Array weekDays after the call to
ChangeArrayElements:");
        DisplayArray(weekDays);
    }
}
// The example displays the following output:
//         Sun Mon Tue Wed Thu Fri Sat
//
//        Array weekDays after the call to ChangeArray:
//        Sat Fri Thu Wed Tue Mon Sun
//
//        Array weekDays after the call to ChangeArrayElements:
//        Mon Wed Fri Wed Tue Mon Sun
```

Multidimensional arrays

Arrays can have more than one dimension. For example, the following declarations create four arrays. Two arrays have two dimensions. Two arrays have three dimensions. The first two declarations declare the length of each dimension, but don't initialize the values of the array. The second two declarations use an initializer to set the values of each element in the multidimensional array.

C#

```
int[,] array2DDeclaration = new int[4, 2];

int[,,] array3DDeclaration = new int[4, 2, 3];

// Two-dimensional array.
int[,] array2DInitialization =  { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } };
// Three-dimensional array.
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                                { { 7, 8, 9 }, { 10, 11, 12 } } };

// Accessing array elements.
System.Console.WriteLine(array2DInitialization[0, 0]);
System.Console.WriteLine(array2DInitialization[0, 1]);
System.Console.WriteLine(array2DInitialization[1, 0]);
System.Console.WriteLine(array2DInitialization[1, 1]);

System.Console.WriteLine(array2DInitialization[3, 0]);
System.Console.WriteLine(array2DInitialization[3, 1]);
// Output:
// 1
// 2
```

```
// 3
// 4
// 7
// 8

System.Console.WriteLine(array3D[1, 0, 1]);
System.Console.WriteLine(array3D[1, 1, 2]);
// Output:
// 8
// 12

// Getting the total count of elements or the length of a given dimension.
var allLength = array3D.Length;
var total = 1;
for (int i = 0; i < array3D.Rank; i++)
{
    total *= array3D.GetLength(i);
}
System.Console.WriteLine($"{allLength} equals {total}");
// Output:
// 12 equals 12
```

For multi-dimensional arrays, elements are traversed such that the indices of the rightmost dimension are incremented first, then the next left dimension, and so on, to the leftmost index. The following example enumerates both a 2D and a 3D array:

C#

```
int[,] numbers2D = { { 9, 99 }, { 3, 33 }, { 5, 55 } };

foreach (int i in numbers2D)
{
    System.Console.Write($"{i} ");
}
// Output: 9 99 3 33 5 55

int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                          { { 7, 8, 9 }, { 10, 11, 12 } } };
foreach (int i in array3D)
{
    System.Console.Write($"{i} ");
}
// Output: 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12
```

In a 2D array, you can think of the left index as the *row* and the right index as the *column*.

However, with multidimensional arrays, using a nested for loop gives you more control over the order in which to process the array elements:

C#

```
int[,,] array3D = new int[,,] { { { 1, 2, 3 }, { 4,   5,  6 } },
                        { { 7, 8, 9 }, { 10, 11, 12 } } };

for (int i = 0; i < array3D.GetLength(0); i++)
{
    for (int j = 0; j < array3D.GetLength(1); j++)
    {
        for (int k = 0; k < array3D.GetLength(2); k++)
        {
            System.Console.Write($"{array3D[i, j, k]} ");
        }
        System.Console.WriteLine();
    }
    System.Console.WriteLine();
}
// Output (including blank lines):
// 1 2 3
// 4 5 6
//
// 7 8 9
// 10 11 12
//
```

Pass multidimensional arrays as arguments

You pass an initialized multidimensional array to a method in the same way that you pass a one-dimensional array. The following code shows a partial declaration of a print method that accepts a two-dimensional array as its argument. You can initialize and pass a new array in one step, as is shown in the following example. In the following example, a two-dimensional array of integers is initialized and passed to the `Print2DArray` method. The method displays the elements of the array.

C#

```
static void Print2DArray(int[,] arr)
{
    // Display the array elements.
    for (int i = 0; i < arr.GetLength(0); i++)
    {
        for (int j = 0; j < arr.GetLength(1); j++)
        {
            System.Console.WriteLine($"Element({i},{j})={arr[i,j]}");
        }
    }
}
static void ExampleUsage()
{
    // Pass the array as an argument.
```

```
    Print2DArray(new int[,] { { 1, 2 }, { 3, 4 }, { 5, 6 }, { 7, 8 } });
}
/* Output:
    Element(0,0)=1
    Element(0,1)=2
    Element(1,0)=3
    Element(1,1)=4
    Element(2,0)=5
    Element(2,1)=6
    Element(3,0)=7
    Element(3,1)=8
*/
```

Jagged arrays

A jagged array is an array whose elements are arrays, possibly of different sizes. A jagged array is sometimes called an "array of arrays." Its elements are reference types and are initialized to `null`. The following examples show how to declare, initialize, and access jagged arrays. The first example, `jaggedArray`, is declared in one statement. Each contained array is created in subsequent statements. The second example, `jaggedArray2` is declared and initialized in one statement. It's possible to mix jagged and multidimensional arrays. The final example, `jaggedArray3`, is a declaration and initialization of a single-dimensional jagged array that contains three two-dimensional array elements of different sizes.

C#

```
int[][] jaggedArray = new int[3][];

jaggedArray[0] = [1, 3, 5, 7, 9];
jaggedArray[1] = [0, 2, 4, 6];
jaggedArray[2] = [11, 22];

int[][] jaggedArray2 =
[
    [1, 3, 5, 7, 9],
    [0, 2, 4, 6],
    [11, 22]
];

// Assign 77 to the second element ([1]) of the first array ([0]):
jaggedArray2[0][1] = 77;

// Assign 88 to the second element ([1]) of the third array ([2]):
jaggedArray2[2][1] = 88;

int[][,] jaggedArray3 =
[
```

```
    new int[,] { {1,3}, {5,7} },
    new int[,] { {0,2}, {4,6}, {8,10} },
    new int[,] { {11,22}, {99,88}, {0,9} }
];

Console.Write("{0}", jaggedArray3[0][1, 0]);
Console.WriteLine(jaggedArray3.Length);
```

A jagged array's elements must be initialized before you can use them. Each of the elements is itself an array. It's also possible to use initializers to fill the array elements with values. When you use initializers, you don't need the array size.

This example builds an array whose elements are themselves arrays. Each one of the array elements has a different size.

C#

```
// Declare the array of two elements.
int[][] arr = new int[2][];

// Initialize the elements.
arr[0] = [1, 3, 5, 7, 9];
arr[1] = [2, 4, 6, 8];

// Display the array elements.
for (int i = 0; i < arr.Length; i++)
{
    System.Console.Write("Element({0}): ", i);

    for (int j = 0; j < arr[i].Length; j++)
    {
        System.Console.Write("{0}{1}", arr[i][j], j == (arr[i].Length - 1) ?
"" : " ");
    }
    System.Console.WriteLine();
}
/* Output:
    Element(0): 1 3 5 7 9
    Element(1): 2 4 6 8
*/
```

Implicitly typed arrays

You can create an implicitly typed array in which the type of the array instance is inferred from the elements specified in the array initializer. The rules for any implicitly typed variable also apply to implicitly typed arrays. For more information, see Implicitly Typed Local Variables.

The following examples show how to create an implicitly typed array:

C#

```
int[] a = new[] { 1, 10, 100, 1000 }; // int[]

// Accessing array
Console.WriteLine("First element: " + a[0]);
Console.WriteLine("Second element: " + a[1]);
Console.WriteLine("Third element: " + a[2]);
Console.WriteLine("Fourth element: " + a[3]);
/* Outputs
First element: 1
Second element: 10
Third element: 100
Fourth element: 1000
*/

var b = new[] { "hello", null, "world" }; // string[]

// Accessing elements of an array using 'string.Join' method
Console.WriteLine(string.Join(" ", b));
/* Output
hello  world
*/

// single-dimension jagged array
int[][] c =
[
    [1,2,3,4],
    [5,6,7,8]
];
// Looping through the outer array
for (int k = 0; k < c.Length; k++)
{
    // Looping through each inner array
    for (int j = 0; j < c[k].Length; j++)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at c[{k}][{j}] is: {c[k][j]}");
    }
}
/* Outputs
Element at c[0][0] is: 1
Element at c[0][1] is: 2
Element at c[0][2] is: 3
Element at c[0][3] is: 4
Element at c[1][0] is: 5
Element at c[1][1] is: 6
Element at c[1][2] is: 7
Element at c[1][3] is: 8
*/

// jagged array of strings
```

```
string[][] d =
[
    ["Luca", "Mads", "Luke", "Dinesh"],
    ["Karen", "Suma", "Frances"]
];

// Looping through the outer array
int i = 0;
foreach (var subArray in d)
{
    // Looping through each inner array
    int j = 0;
    foreach (var element in subArray)
    {
        // Accessing each element and printing it to the console
        Console.WriteLine($"Element at d[{i}][{j}] is: {element}");
        j++;
    }
    i++;
}
/* Outputs
Element at d[0][0] is: Luca
Element at d[0][1] is: Mads
Element at d[0][2] is: Luke
Element at d[0][3] is: Dinesh
Element at d[1][0] is: Karen
Element at d[1][1] is: Suma
Element at d[1][2] is: Frances
*/
```

In the previous example, notice that with implicitly typed arrays, no square brackets are used on the left side of the initialization statement. Also, jagged arrays are initialized by using `new []` just like single-dimensional arrays.

When you create an anonymous type that contains an array, the array must be implicitly typed in the type's object initializer. In the following example, `contacts` is an implicitly typed array of anonymous types, each of which contains an array named `PhoneNumbers`. The `var` keyword isn't used inside the object initializers.

C#

```
var contacts = new[]
{
    new
    {
        Name = "Eugene Zabokritski",
        PhoneNumbers = new[] { "206-555-0108", "425-555-0001" }
    },
    new
    {
        Name = "Hanying Feng",
```

```
        PhoneNumbers = new[] { "650-555-0199" }
    }
};
```

void (C# reference)

Article • 03/30/2024

You use `void` as the return type of a method (or a local function) to specify that the method doesn't return a value.

C#

```
public static void Display(IEnumerable<int> numbers)
{
    if (numbers is null)
    {
        return;
    }

    Console.WriteLine(string.Join(" ", numbers));
}
```

You can also use `void` as a referent type to declare a pointer to an unknown type. For more information, see Pointer types.

You cannot use `void` as the type of a variable.

See also

- System.Void

Unmanaged types (C# reference)

Article • 04/02/2024

A type is an **unmanaged type** if it's any of the following types:

- `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `long`, `ulong`, `nint`, `nuint`, `char`, `float`, `double`, `decimal`, or `bool`
- Any enum type
- Any pointer type
- A tuple whose members are all of an unmanaged type
- Any user-defined struct type that contains fields of unmanaged types only.

You can use the unmanaged constraint to specify that a type parameter is a non-pointer, non-nullable unmanaged type.

A *constructed* struct type that contains fields of unmanaged types only is also unmanaged, as the following example shows:

C#

```
using System;

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

public class UnmanagedTypes
{
    public static void Main()
    {
        DisplaySize<Coords<int>>();
        DisplaySize<Coords<double>>();
    }

    private unsafe static void DisplaySize<T>() where T : unmanaged
    {
        Console.WriteLine($"{typeof(T)} is unmanaged and its size is
{sizeof(T)} bytes");
    }
}
// Output:
// Coords`1[System.Int32] is unmanaged and its size is 8 bytes
// Coords`1[System.Double] is unmanaged and its size is 16 bytes
```

A generic struct may be the source of both unmanaged and managed constructed types. The preceding example defines a generic struct `Coords<T>` and presents the examples of unmanaged constructed types. The example of a managed type is `Coords<object>`. It's managed because it has the fields of the `object` type, which is managed. If you want *all* constructed types to be unmanaged types, use the `unmanaged` constraint in the definition of a generic struct:

```
C#

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

C# language specification

For more information, see the Pointer types section of the C# language specification.

See also

- Pointer types
- Memory and span-related types
- sizeof operator
- stackalloc

Default values of C# types (C# reference)

Article • 12/04/2024

The following table shows the default values of C# types:

Expand table

Type	Default value
Any reference type	`null`
Any built-in integral numeric type	0 (zero)
Any built-in floating-point numeric type	0 (zero)
bool	`false`
char	`'\0'` (U+0000)
enum	The value produced by the expression `(E)0`, where `E` is the enum identifier.
struct	The value produced by setting all value-type fields to their default values and all reference-type fields to `null`.
Any nullable value type	An instance for which the HasValue property is `false` and the Value property is undefined. That default value is also known as the *null* value of a nullable value type.

Default value expressions

Use the default operator to produce the default value of a type, as the following example shows:

```
C#
```

```
int a = default(int);
```

You can use the default literal to initialize a variable with the default value of its type:

```
C#
```

```
int a = default;
```

Parameterless constructor of a value type

For a value type, the *implicit* parameterless constructor also produces the default value of the type, as the following example shows:

C#

```
var n = new System.Numerics.Complex();
Console.WriteLine(n);  // output: (0, 0)
```

At run time, if the System.Type instance represents a value type, you can use the Activator.CreateInstance(Type) method to invoke the parameterless constructor to obtain the default value of the type.

> **Note**
>
> A **structure type** (which is a value type) may have an **explicit parameterless constructor** that may produce a non-default value of the type. Thus, we recommend using the `default` operator or the `default` literal to produce the default value of a type.

C# language specification

For more information, see the following sections of the C# language specification:

- Default values
- Default constructors
- Parameterless struct constructors
- C# 11 - Auto default structs

See also

- Constructors

C# Keywords

Article • 04/17/2025

Keywords are predefined, reserved identifiers that have special meanings to the compiler. They can't be used as identifiers in your program unless they include `@` as a prefix. For example, `@if` is a valid identifier, but `if` isn't because `if` is a keyword.

The first table in this article lists keywords that are reserved identifiers in any part of a C# program. The second table in this article lists the contextual keywords in C#. Contextual keywords have special meaning only in a limited program context and can be used as identifiers outside that context. Generally, as new keywords are added to the C# language, they're added as contextual keywords in order to avoid breaking programs written in earlier versions.

abstract
as
base
bool
break
byte
case
catch
char
checked
class
const
continue
decimal
default
delegate
do
double
else
enum

event
explicit
extern
false
finally
fixed

float
for
foreach
goto
if
implicit
in
int
interface
internal
is
lock
long

namespace
new
null
object
operator
out
override
params
private
protected
public
readonly
ref
return
sbyte
sealed
short
sizeof
stackalloc

static
string
struct
switch
this
throw
true

try
typeof
uint
ulong
unchecked
unsafe
ushort
using
virtual
void
volatile
while

Contextual keywords

A contextual keyword is used to provide a specific meaning in the code, but it isn't a reserved word in C#. Some contextual keywords, such as `partial` and `where`, have special meanings in two or more contexts.

add
allows
alias
and
ascending
args
async
await
by
descending
dynamic
equals
extension

field
file
from
get
global
group
init

into
join
let
managed (function pointer calling convention)
nameof

nint
not
notnull
nuint
on
or
orderby
partial (type)
partial (member)
record
remove
required

scoped
select
set
unmanaged (function pointer calling convention)
unmanaged (generic type constraint)
value
var
when (filter condition)
where (generic type constraint)
where (query clause)
with
yield

Access Modifiers (C# Reference)

Article • 10/22/2024

Access modifiers are keywords used to specify the declared accessibility of a member or a type. This section introduces the five access modifiers:

- `public`
- `protected`
- `internal`
- `private`
- `file`

The following seven accessibility levels can be specified using the access modifiers:

- public: Access isn't restricted.
- protected: Access is limited to the containing class or types derived from the containing class.
- internal: Access is limited to the current assembly.
- protected internal: Access is limited to the current assembly or types derived from the containing class.
- private: Access is limited to the containing type.
- private protected: Access is limited to the containing class or types derived from the containing class within the current assembly.
- file: The declared type is only visible in the current source file. File scoped types are generally used for source generators.

This section also introduces the following concepts:

- Accessibility Levels: Using the access modifiers to declare levels of accessibility.
- Accessibility Domain: Specifies where, in the program sections, a member can be referenced.
- Restrictions on Using Accessibility Levels: A summary of the restrictions on using declared accessibility levels.

See also

- Add accessibility modifiers (style rule IDE0040)
- C# Keywords
- Access Modifiers
- Access Keywords
- Modifiers

Accessibility Levels (C# Reference)

Article • 08/16/2023

Use the access modifiers, `public`, `protected`, `internal`, or `private`, to specify one of the following declared accessibility levels for members.

Expand table

Declared accessibility	Meaning
public	Access is not restricted.
protected	Access is limited to the containing class or types derived from the containing class.
internal	Access is limited to the current assembly.
protected internal	Access is limited to the current assembly or types derived from the containing class.
private	Access is limited to the containing type.
private protected	Access is limited to the containing class or types derived from the containing class within the current assembly.

Only one access modifier is allowed for a member or type, except when you use the `protected internal` or `private protected` combinations.

Access modifiers are not allowed on namespaces. Namespaces have no access restrictions.

Depending on the context in which a member declaration occurs, only certain declared accessibilities are permitted. If no access modifier is specified in a member declaration, a default accessibility is used.

Top-level types, which are not nested in other types, can only have `internal` or `public` accessibility. The default accessibility for these types is `internal`.

Nested types, which are members of other types, can have declared accessibilities as indicated in the following table.

Expand table

Members of	Default member accessibility	Allowed declared accessibility of the member
`enum`	`public`	None

`class`	`private`	`public`
		`protected`
		`internal`
		`private`
		`protected internal`
		`private protected`

`interface`	`public`	`public`
		`protected`
		`internal`
		`private`*
		`protected internal`
		`private protected`

`struct`	`private`	`public`
		`internal`
		`private`

* An `interface` member with `private` accessibility must have a default implementation.

> **Note**
>
> If a class or struct is modified with the record keyword modifier, then the same access modifiers are allowed.
> Also, with the record modifier the default member accessibility is still `private` both for class and the struct.

The accessibility of a nested type depends on its accessibility domain, which is determined by both the declared accessibility of the member and the accessibility

domain of the immediately containing type. However, the accessibility domain of a nested type cannot exceed that of the containing type.

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

- C# Keywords
- Access Modifiers
- Accessibility Domain
- Restrictions on Using Accessibility Levels
- Access Modifiers
- public
- private
- protected
- internal

Accessibility Domain (C# Reference)

Article • 03/12/2024

The accessibility domain of a member specifies in which program sections a member can be referenced. If the member is nested within another type, its accessibility domain is determined by both the accessibility level of the member and the accessibility domain of the immediately containing type.

The accessibility domain of a top-level type is at least the program text of the project that it is declared in. That is, the domain includes all of the source files of this project. The accessibility domain of a nested type is at least the program text of the type in which it is declared. That is, the domain is the type body, which includes all nested types. The accessibility domain of a nested type never exceeds that of the containing type. These concepts are demonstrated in the following example.

Example

This example contains a top-level type, `T1`, and two nested classes, `M1` and `M2`. The classes contain fields that have different declared accessibilities. In the `Main` method, a comment follows each statement to indicate the accessibility domain of each member. Notice that the statements that try to reference the inaccessible members are commented out. If you want to see the compiler errors caused by referencing an inaccessible member, remove the comments one at a time.

C#

```
public class T1
{
    public static int publicInt;
    internal static int internalInt;
    private static int privateInt = 0;

    static T1()
    {
        // T1 can access public or internal members
        // in a public or private (or internal) nested class.
        M1.publicInt = 1;
        M1.internalInt = 2;
        M2.publicInt = 3;
        M2.internalInt = 4;

        // Cannot access the private member privateInt
        // in either class:
        // M1.privateInt = 2; //CS0122
    }
```

```
    public class M1
    {
        public static int publicInt;
        internal static int internalInt;
        private static int privateInt = 0;
    }

    private class M2
    {
        public static int publicInt = 0;
        internal static int internalInt = 0;
        private static int privateInt = 0;
    }
}

class MainClass
{
    static void Main()
    {
        // Access is unlimited.
        T1.publicInt = 1;

        // Accessible only in current assembly.
        T1.internalInt = 2;

        // Error CS0122: inaccessible outside T1.
        // T1.privateInt = 3;

        // Access is unlimited.
        T1.M1.publicInt = 1;

        // Accessible only in current assembly.
        T1.M1.internalInt = 2;

        // Error CS0122: inaccessible outside M1.
        //    T1.M1.privateInt = 3;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.publicInt = 1;

        // Error CS0122: inaccessible outside T1.
        //    T1.M2.internalInt = 2;

        // Error CS0122: inaccessible outside M2.
        //    T1.M2.privateInt = 3;

        // Keep the console open in debug mode.
        System.Console.WriteLine("Press any key to exit.");
        System.Console.ReadKey();
    }
}
```

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

- C# Keywords
- Access Modifiers
- Accessibility Levels
- Restrictions on Using Accessibility Levels
- Access Modifiers
- public
- private
- protected
- internal

Restrictions on using accessibility levels (C# Reference)

Article • 03/12/2024

When you specify a type in a declaration, check whether the accessibility level of the type is dependent on the accessibility level of a member or of another type. For example, the direct base class must be at least as accessible as the derived class. The following declarations cause a compiler error because the base class `BaseClass` is less accessible than `MyClass`:

C#

```
class BaseClass {...}
public class MyClass: BaseClass {...} // Error
```

The following table summarizes the restrictions on declared accessibility levels.

Expand table

Context	Remarks
Classes	The direct base class of a class type must be at least as accessible as the class type itself.
Interfaces	The explicit base interfaces of an interface type must be at least as accessible as the interface type itself.
Delegates	The return type and parameter types of a delegate type must be at least as accessible as the delegate type itself.
Constants	The type of a constant must be at least as accessible as the constant itself.
Fields	The type of a field must be at least as accessible as the field itself.
Methods	The return type and parameter types of a method must be at least as accessible as the method itself.
Properties	The type of a property must be at least as accessible as the property itself.
Events	The type of an event must be at least as accessible as the event itself.
Indexers	The type and parameter types of an indexer must be at least as accessible as the indexer itself.
Operators	The return type and parameter types of an operator must be at least as accessible as the operator itself.

Context	Remarks
Constructors	The parameter types of a constructor must be at least as accessible as the constructor itself.

Example

The following example contains erroneous declarations of different types. The comment following each declaration indicates the expected compiler error.

C#

```
// Restrictions on Using Accessibility Levels
// CS0052 expected as well as CS0053, CS0056, and CS0057
// To make the program work, change access level of both class B
// and MyPrivateMethod() to public.

using System;

// A delegate:
delegate int MyDelegate();

class B
{
    // A private method:
    static int MyPrivateMethod()
    {
        return 0;
    }
}

public class A
{
    // Error: The type B is less accessible than the field A.myField.
    public B myField = new B();

    // Error: The type B is less accessible
    // than the constant A.myConst.
    public readonly B myConst = new B();

    public B MyMethod()
    {
        // Error: The type B is less accessible
        // than the method A.MyMethod.
        return new B();
    }

    // Error: The type B is less accessible than the property A.MyProp
    public B MyProp
    {
        set
```

```
        {
        }
    }

    MyDelegate d = new MyDelegate(B.MyPrivateMethod);
    // Even when B is declared public, you still get the error:
    // "The parameter B.MyPrivateMethod is not accessible due to
    // protection level."

    public static B operator +(A m1, B m2)
    {
        // Error: The type B is less accessible
        // than the operator A.operator +(A,B)
        return new B();
    }

    static void Main()
    {
        Console.Write("Compiled successfully");
    }
}
```

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

- C# Keywords
- Access Modifiers
- Accessibility Domain
- Accessibility Levels
- Access Modifiers
- public
- private
- protected
- internal

internal (C# Reference)

Article • 03/30/2024

The `internal` keyword is an access modifier for types and type members.

> This page covers `internal` access. The `internal` keyword is also part of the protected internal access modifier.

Internal types or members are accessible only within files in the same assembly, as in this example:

```
C#
```

```
public class BaseClass
{
    // Only accessible within the same assembly.
    internal static int x = 0;
}
```

For a comparison of `internal` with the other access modifiers, see Accessibility Levels and Access Modifiers.

An `assembly` is an executable or dynamic link library (DLL) produced from compiling one or more source files.

For more information about assemblies, see Assemblies in .NET.

A common use of internal access is in component-based development because it enables a group of components to cooperate in a private manner without being exposed to the rest of the application code. For example, a framework for building graphical user interfaces could provide `Control` and `Form` classes that cooperate by using members with internal access. Since these members are internal, they are not exposed to code that is using the framework.

It is an error to reference a type or a member with internal access outside the assembly within which it was defined.

Example 1

This example contains two files, `Assembly1.cs` and `Assembly1_a.cs`. The first file contains an internal base class, `BaseClass`. In the second file, an attempt to instantiate `BaseClass` will produce an error.

C#

```
// Assembly1.cs
// Compile with: /target:library
internal class BaseClass
{
   public static int intM = 0;
}
```

C#

```
// Assembly1_a.cs
// Compile with: /reference:Assembly1.dll
class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // CS0122
   }
}
```

Example 2

In this example, use the same files you used in example 1, and change the accessibility level of `BaseClass` to `public`. Also change the accessibility level of the member `intM` to `internal`. In this case, you can instantiate the class, but you cannot access the internal member.

C#

```
// Assembly2.cs
// Compile with: /target:library
public class BaseClass
{
   internal static int intM = 0;
}
```

C#

```
// Assembly2_a.cs
// Compile with: /reference:Assembly2.dll
public class TestAccess
{
   static void Main()
   {
      var myBase = new BaseClass();   // Ok.
      BaseClass.intM = 444;    // CS0117
```

```
    }
}
```

C# Language Specification

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

See also

- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- protected

private (C# Reference)

Article • 03/30/2024

The `private` keyword is a member access modifier.

> This page covers `private` access. The `private` keyword is also part of the private protected access modifier.

Private access is the least permissive access level. Private members are accessible only within the body of the class or the struct in which they are declared, as in this example:

C#

```
class Employee
{
    private int _i;
    double _d;   // private access by default
}
```

Nested types in the same body can also access those private members.

It is a compile-time error to reference a private member outside the class or the struct in which it is declared.

For a comparison of `private` with the other access modifiers, see Accessibility Levels and Access Modifiers.

Example

In this example, the `Employee` class contains two private data members, `_name` and `_salary`. As private members, they cannot be accessed except by member methods. Public methods named `GetName` and `Salary` are added to allow controlled access to the private members. The `_name` member is accessed by way of a public method, and the `_salary` member is accessed by way of a public read-only property. For more information, see Properties.

C#

```
class Employee2
{
    private readonly string _name = "FirstName, LastName";
    private readonly double _salary = 100.0;
```

```
    public string GetName()
    {
        return _name;
    }

    public double Salary
    {
        get { return _salary; }
    }
}

class PrivateTest
{
    static void Main()
    {
        var e = new Employee2();

        // The data members are inaccessible (private), so
        // they can't be accessed like this:
        //    string n = e._name;
        //    double s = e._salary;

        // '_name' is indirectly accessed via method:
        string n = e.GetName();

        // '_salary' is indirectly accessed via property
        double s = e.Salary;
    }
}
```

C# language specification

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

See also

- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- protected
- internal

protected (C# Reference)

07/16/2025

The `protected` keyword is a member access modifier.

> **Note**
>
> This page covers `protected` access. The `protected` keyword is also part of the **protected internal** and **private protected** access modifiers.

A protected member is accessible within its class and by derived class instances.

For a comparison of `protected` with the other access modifiers, see Accessibility Levels.

Example 1

A protected member of a base class is accessible in a derived class only if the access occurs through the derived class type. For example, consider the following code segment:

C#

```
namespace Example1
{
    class BaseClass
    {
        protected int myValue = 123;
    }

    class DerivedClass : BaseClass
    {
        static void Main()
        {
            var baseObject = new BaseClass();
            var derivedObject = new DerivedClass();

            // Error CS1540, because myValue can only be accessed through
            // the derived class type, not through the base class type.
            // baseObject.myValue = 10;

            // OK, because this class derives from BaseClass.
            derivedObject.myValue = 10;
        }
    }
}
```

The statement `baseObject.myValue = 10` generates an error because it accesses the protected member through a base class reference (`baseObject` is of type `BaseClass`). Protected members can only be accessed through the derived class type or types derived from it.

Unlike `private protected`, the `protected` access modifier allows access from derived classes **in any assembly**. Unlike `protected internal`, it does **not** allow access from non-derived classes within the same assembly.

Struct members cannot be protected because the struct cannot be inherited.

Example 2

In this example, the class `DerivedPoint` is derived from `Point`. Therefore, you can access the protected members of the base class directly from the derived class.

C#

```
namespace Example2
{
    class Point
    {
        protected int x;
        protected int y;
    }

    class DerivedPoint: Point
    {
        static void Main()
        {
            var dpoint = new DerivedPoint();

            // Direct access to protected members.
            dpoint.x = 10;
            dpoint.y = 15;
            Console.WriteLine($"x = {dpoint.x}, y = {dpoint.y}");
        }
    }
    // Output: x = 10, y = 15
}
```

If you change the access levels of `x` and `y` to private, the compiler will issue the error messages:

`'Point.y' is inaccessible due to its protection level.`

`'Point.x' is inaccessible due to its protection level.`

Cross-assembly access

The following example demonstrates that `protected` members are accessible from derived classes even when they're in different assemblies:

C#

```
// Assembly1.cs
// Compile with: /target:library
namespace Assembly1
{
    public class BaseClass
    {
        protected int myValue = 0;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
namespace Assembly2
{
    using Assembly1;

    class DerivedClass : BaseClass
    {
        void Access()
        {
            // OK, because protected members are accessible from
            // derived classes in any assembly
            myValue = 10;
        }
    }
}
```

This cross-assembly accessibility is what distinguishes `protected` from `private protected` (which restricts access to the same assembly) but is similar to `protected internal` (though `protected internal` also allows same-assembly access from non-derived classes).

C# language specification

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

See also

- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- internal
- Security concerns for internal virtual keywords

public (C# Reference)

Article • 03/12/2024

The `public` keyword is an access modifier for types and type members. Public access is the most permissive access level. There are no restrictions on accessing public members, as in this example:

C#

```
class SampleClass
{
    public int x; // No access restrictions.
}
```

See Access Modifiers and Accessibility Levels for more information.

Example

In the following example, two classes are declared, `PointTest` and `Program`. The public members `x` and `y` of `PointTest` are accessed directly from `Program`.

C#

```
class PointTest
{
    public int x;
    public int y;
}

class Program
{
    static void Main()
    {
        var p = new PointTest();
        // Direct access to public members.
        p.x = 10;
        p.y = 15;
        Console.WriteLine($"x = {p.x}, y = {p.y}");
    }
}
// Output: x = 10, y = 15
```

If you change the `public` access level to private or protected, you will get the error message:

'PointTest.y' is inaccessible due to its protection level.

C# language specification

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

See also

- Access Modifiers
- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- private
- protected
- internal

protected internal (C# Reference)

07/16/2025

The `protected internal` keyword combination is a member access modifier. A protected internal member is accessible from the current assembly or from types that are derived from the containing class. For a comparison of `protected internal` with the other access modifiers, see Accessibility Levels.

Example

A protected internal member of a base class is accessible from any type within its containing assembly. It is also accessible in a derived class located in another assembly only if the access occurs through a variable of the derived class type. For example, consider the following code segment:

C#

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
   protected internal int myValue = 0;
}

class TestAccess
{
    void Access()
    {
        var baseObject = new BaseClass();
        baseObject.myValue = 5;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass : BaseClass
{
    static void Main()
    {
        var baseObject = new BaseClass();
        var derivedObject = new DerivedClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 10;
```

```
        // OK, because this class derives from BaseClass.
        derivedObject.myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and another class, `TestAccess`. `BaseClass` owns a protected internal member, `myValue`, which is accessed by the `TestAccess` type because they're in the same assembly. In the second file, an attempt to access `myValue` through an instance of `BaseClass` will produce an error, while an access to this member through an instance of a derived class, `DerivedClass` will succeed. This shows that `protected internal` allows access from **any class within the same assembly** or **derived classes in any assembly**, making it the most permissive of the protected access modifiers.

Struct members cannot be `protected internal` because the struct cannot be inherited.

Overriding protected internal members

When overriding a virtual member, the accessibility modifier of the overridden method depends on the assembly where the derived class is defined.

When the derived class is defined in the same assembly as the base class, all overridden members have `protected internal` access. If the derived class is defined in a different assembly from the base class, overridden members have `protected` access.

C#

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    protected internal virtual int GetExampleValue()
    {
        return 5;
    }
}

public class DerivedClassSameAssembly : BaseClass
{
    // Override to return a different example value, accessibility modifiers
remain the same.
    protected internal override int GetExampleValue()
    {
        return 9;
```

```
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClassDifferentAssembly : BaseClass
{
    // Override to return a different example value, since this override
    // method is defined in another assembly, the accessibility modifiers
    // are only protected, instead of protected internal.
    protected override int GetExampleValue()
    {
        return 2;
    }
}
```

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

- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- internal
- Security concerns for internal virtual keywords

private protected (C# Reference)

07/16/2025

The `private protected` keyword combination is a member access modifier. A private protected member is accessible by types derived from the containing class, but only within its containing assembly. For a comparison of `private protected` with the other access modifiers, see Accessibility Levels.

> **Note**
>
> The `private protected` access modifier is valid in C# version 7.2 and later.

Example

A private protected member of a base class is accessible from derived types in its containing assembly only if the static type of the variable is the derived class type. For example, consider the following code segment:

C#

```
// Assembly1.cs
// Compile with: /target:library
public class BaseClass
{
    private protected int myValue = 0;
}

public class DerivedClass1 : BaseClass
{
    void Access()
    {
        var baseObject = new BaseClass();

        // Error CS1540, because myValue can only be accessed by
        // classes derived from BaseClass.
        // baseObject.myValue = 5;

        // OK, accessed through the current derived class instance
        myValue = 5;
    }
}
```

C#

```
// Assembly2.cs
// Compile with: /reference:Assembly1.dll
class DerivedClass2 : BaseClass
{
    void Access()
    {
        // Error CS0122, because myValue can only be
        // accessed by types in Assembly1
        // myValue = 10;
    }
}
```

This example contains two files, `Assembly1.cs` and `Assembly2.cs`. The first file contains a public base class, `BaseClass`, and a type derived from it, `DerivedClass1`. `BaseClass` owns a private protected member, `myValue`, which `DerivedClass1` can access as an inherited member within the same assembly.

In the second file, an attempt to access `myValue` as an inherited member of `DerivedClass2` will produce an error, because `private protected` members are only accessible by derived types **within the same assembly**. This is the key difference from `protected` (which allows access from derived classes in any assembly) and `protected internal` (which allows access from any class within the same assembly or derived classes in any assembly).

If `Assembly1.cs` contains an InternalsVisibleToAttribute that names `Assembly2`, the derived class `DerivedClass2` will have access to `private protected` members declared in `BaseClass`. `InternalsVisibleTo` makes `private protected` members visible to derived classes in other assemblies.

Comparison with other protected access modifiers

The following table summarizes the key differences between the three protected access modifiers:

Expand table

Access Modifier	Same Assembly, Derived Class	Same Assembly, Non-derived Class	Different Assembly, Derived Class
`protected`	✔	✘	✔
`protected internal`	✔	✔	✔

Access Modifier	Same Assembly, Derived Class	Same Assembly, Non-derived Class	Different Assembly, Derived Class
`private protected`	✔	✘	✘

- Use `protected` when you want derived classes in any assembly to access the member
- Use `protected internal` when you want the most permissive access (any class in same assembly OR derived classes anywhere)
- Use `private protected` when you want the most restrictive protected access (only derived classes in the same assembly)

Struct members cannot be `private protected` because the struct cannot be inherited.

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

- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- private
- internal
- Security concerns for internal virtual keywords

The file modifier

Article • 12/14/2024

Beginning with C# 11, the `file` contextual keyword is a type modifier.

The `file` modifier restricts a top-level type's visibility to the file in which it's declared. The `file` modifier is most often applied to types written by a source generator. File-local types provide source generators with a convenient way to avoid name collisions among generated types. The `file` modifier declares a file-local type, as in this example:

C#

```
file class HiddenWidget
{
    // implementation
}
```

Any types nested within a file-local type are also only visible within the file in which it's declared. Other types in an assembly can use the same name as a file-local type. Because the file-local type is visible only in the file where it's declared, these types don't create a naming collision.

A file-local type can't be the return type or parameter type of any member declared in a non file-local type. A file-local type can't be a field member of a non-file-local. However, a more visible type can implicitly implement a file-local interface type. The type can also explicitly implement a file-local interface but explicit implementations can only be used within the same file.

The following example shows a public type that uses a file-local type to provide a worker method. In addition, the public type implements a file-local interface implicitly:

C#

```
// In File1.cs:
file interface IWidget
{
    int ProvideAnswer();
}

file class HiddenWidget
{
    public int Work() => 42;
}

public class Widget : IWidget
```

```
{
    public int ProvideAnswer()
    {
        var worker = new HiddenWidget();
        return worker.Work();
    }
}
```

In another source file, you can declare types that have the same names as the file-local types. The file-local types aren't visible:

C#

```
// In File2.cs:
// Doesn't conflict with HiddenWidget
// declared in File1.cs
public class HiddenWidget
{
    public void RunTask()
    {
        // omitted
    }
}
```

Member lookup prefers a file-local type declared in the same file over a non-file-local type declared in a different file. This rule ensures that a source generator can rely on member lookup resolving to a file-local type without ambiguity with other type declarations. In the preceding example, all uses of `HiddenWidget` in *File1.cs* resolve to the file-local type declared in *File1.cs*. The file-local declaration of `HiddenWidget` hides the public declaration in *File2.cs*.

C# language specification

For more information, see Declared accessibility in the C# Language Specification, and the C# 11 - File local types feature specification.

See also

- C# Keywords
- Access Modifiers
- Accessibility Levels
- Modifiers
- public
- protected

- internal

abstract (C# Reference)

08/20/2025

The `abstract` modifier indicates that the thing being modified has a missing or incomplete implementation. The abstract modifier can be used with classes, methods, properties, indexers, and events. Use the `abstract` modifier in a class declaration to indicate that a class is intended only to be a base class of other classes, not instantiated on its own. Members marked as abstract must be implemented by non-abstract classes that derive from the abstract class.

Abstract classes can contain both abstract members (which have no implementation and must be overridden in derived classes) and fully implemented members (such as regular methods, properties, and constructors). This allows abstract classes to provide common functionality while still requiring derived classes to implement specific abstract members.

Example 1 - Abstract class with mixed members

The following example demonstrates an abstract class that contains both implemented methods and abstract members:

C#

```
namespace LanguageKeywords;

public abstract class Vehicle
{
    protected string _brand;

    // Constructor - implemented method in abstract class
    public Vehicle(string brand) => _brand = brand;

    // Implemented method - provides functionality that all vehicles share
    public string GetInfo() => $"This is a {_brand} vehicle.";

    // Another implemented method
    public virtual void StartEngine() => Console.WriteLine($"{_brand} engine is
starting...");

    // Abstract method - must be implemented by derived classes
    public abstract void Move();

    // Abstract property - must be implemented by derived classes
    public abstract int MaxSpeed { get; }
}

public class Car : Vehicle
{
    public Car(string brand) : base(brand) { }
```

```
    // Implementation of abstract method
    public override void Move() => Console.WriteLine($"{_brand} car is driving on
the road.");

    // Implementation of abstract property
    public override int MaxSpeed => 200;
}

public class Boat : Vehicle
{
    public Boat(string brand) : base(brand) { }

    // Implementation of abstract method
    public override void Move() => Console.WriteLine($"{_brand} boat is sailing on
the water.");

    // Implementation of abstract property
    public override int MaxSpeed => 50;
}

public class AbstractExample
{
    public static void Examples()
    {
        // Cannot instantiate abstract class: Vehicle v = new Vehicle("Generic");
// Error!

        Car car = new Car("Toyota");
        Boat boat = new Boat("Yamaha");

        // Using implemented methods from abstract class
        Console.WriteLine(car.GetInfo());
        car.StartEngine();

        // Using abstract methods implemented in derived class
        car.Move();
        Console.WriteLine($"Max speed: {car.MaxSpeed} km/h");

        Console.WriteLine();

        Console.WriteLine(boat.GetInfo());
        boat.StartEngine();
        boat.Move();
        Console.WriteLine($"Max speed: {boat.MaxSpeed} km/h");
    }
}

class Program
{
    static void Main()
    {
        AbstractExample.Examples();
    }
}
```

```
/* Output:
This is a Toyota vehicle.
Toyota engine is starting...
Toyota car is driving on the road.
Max speed: 200 km/h

This is a Yamaha vehicle.
Yamaha engine is starting...
Yamaha boat is sailing on the water.
Max speed: 50 km/h
*/
```

In this example, the `Vehicle` abstract class provides:

- **Implemented members**: `GetInfo()` method, `StartEngine()` method, and constructor - these provide common functionality for all vehicles.
- **Abstract members**: `Move()` method and `MaxSpeed` property - these must be implemented by each specific vehicle type.

This design allows the abstract class to provide shared functionality while ensuring that derived classes implement vehicle-specific behavior.

Example 2

In this example, the class `Square` must provide an implementation of `GetArea` because it derives from `Shape`:

C#

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
```

```
}
// Output: Area of the square = 144
```

Abstract classes have the following features:

- An abstract class cannot be instantiated.
- An abstract class may contain abstract methods and accessors.
- An abstract class can also contain implemented methods, properties, fields, and other members that provide functionality to derived classes.
- It is not possible to modify an abstract class with the sealed modifier because the two modifiers have opposite meanings. The `sealed` modifier prevents a class from being inherited and the `abstract` modifier requires a class to be inherited.
- A non-abstract class derived from an abstract class must include actual implementations of all inherited abstract methods and accessors.

Use the `abstract` modifier in a method or property declaration to indicate that the method or property does not contain implementation.

Abstract methods have the following features:

- An abstract method is implicitly a virtual method.
- Abstract method declarations are only permitted in abstract classes.
- Because an abstract method declaration provides no actual implementation, there is no method body; the method declaration simply ends with a semicolon and there are no curly braces ({ }) following the signature. For example:

  C#

  ```
  public abstract void MyMethod();
  ```

  The implementation is provided by a method override, which is a member of a non-abstract class.
- It is an error to use the static or virtual modifiers in an abstract method declaration.

Abstract properties behave like abstract methods, except for the differences in declaration and invocation syntax.

- It is an error to use the `abstract` modifier on a static property.

- An abstract inherited property can be overridden in a derived class by including a property declaration that uses the override modifier.

For more information about abstract classes, see Abstract and Sealed Classes and Class Members.

An abstract class must provide implementation for all interface members.

An abstract class that implements an interface might map the interface methods onto abstract methods. For example:

C#

```
interface I
{
    void M();
}

abstract class C : I
{
    public abstract void M();
}
```

Example 3

In this example, the class `DerivedClass` is derived from an abstract class `BaseClass`. The abstract class contains an abstract method, `AbstractMethod`, and two abstract properties, `X` and `Y`.

C#

```
// Abstract class
abstract class BaseClass
{
    protected int _x = 100;
    protected int _y = 150;

    // Abstract method
    public abstract void AbstractMethod();

    // Abstract properties
    public abstract int X { get; }
    public abstract int Y { get; }
}

class DerivedClass : BaseClass
{
    public override void AbstractMethod()
```

```
    {
        _x++;
        _y++;
    }

    public override int X   // overriding property
    {
        get
        {
            return _x + 10;
        }
    }

    public override int Y   // overriding property
    {
        get
        {
            return _y + 10;
        }
    }

    static void Main()
    {
        var o = new DerivedClass();
        o.AbstractMethod();
        Console.WriteLine($"x = {o.X}, y = {o.Y}");
    }
}
// Output: x = 111, y = 161
```

In the preceding example, if you attempt to instantiate the abstract class by using a statement like this:

C#

```
BaseClass bc = new BaseClass();   // Error
```

You will get an error saying that the compiler cannot create an instance of the abstract class 'BaseClass'.

Nonetheless, it is possible to use an abstract class constructor, as in the example below

Example 4

C#

```
public abstract class Shape
{
    public string Color { get; set; }
```

```
    // Constructor of the abstract class
    protected Shape(string color)
    {
        Color = color;
        Console.WriteLine($"Created a shape with color {color}.");
    }

    // Abstract method that must be implemented by derived classes
    public abstract double CalculateArea();
}

public class Square : Shape
{
    public double Side { get; set; }

    // Constructor of the derived class calling the base class constructor
    public Square(string color, double side) : base(color)
    {
        Side = side;
    }

    public override double CalculateArea()
    {
        return Side * Side;
    }
}

public class Program
{
    public static void Main(string[] args)
     {
            Square square = new Square("red", 5);
            Console.WriteLine($"Area of the square: {square.CalculateArea()}");
     }
}
```

The `Shape` class is declared `abstract`, which means it cannot be instantiated directly. Instead, it serves as a blueprint for other classes.

- Even though you can't create objects of an abstract class, it can still have a constructor. This constructor is typically `protected`, meaning it can only be accessed from derived classes. In this case, the `Shape` constructor takes a `color` parameter and initializes the `Color` property. It also prints a message to the console. The `public Square(string color, double side) : base(color)` part calls the base class's constructor (`Shape`) and passes the `color` argument to it.
- In the Shape class, the defined constructor takes a color as a parameter `protected Shape(string color)`. This means there's no longer a default parameterless constructor automatically provided by C# thus derived classes must use the `: base(color)` expression

to invoke the base constructor. Setting the default value to color `protected Shape(string color="green")` will allow to omit the `: base(color)` expression in derived classes, still such constructor `protected Shape(string color="green")` will be invoked, setting the color to green.

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

- virtual
- override
- C# Keywords

async (C# Reference)

Article • 03/22/2023

Use the `async` modifier to specify that a method, lambda expression, or anonymous method is asynchronous. If you use this modifier on a method or expression, it's referred to as an *async method*. The following example defines an async method named `ExampleMethodAsync`:

```
C#

public async Task<int> ExampleMethodAsync()
{
    //...
}
```

If you're new to asynchronous programming or do not understand how an async method uses the await operator to do potentially long-running work without blocking the caller's thread, read the introduction in Asynchronous programming with async and await. The following code is found inside an async method and calls the HttpClient.GetStringAsync method:

```
C#

string contents = await httpClient.GetStringAsync(requestUrl);
```

An async method runs synchronously until it reaches its first `await` expression, at which point the method is suspended until the awaited task is complete. In the meantime, control returns to the caller of the method, as the example in the next section shows.

If the method that the `async` keyword modifies doesn't contain an `await` expression or statement, the method executes synchronously. A compiler warning alerts you to any async methods that don't contain `await` statements, because that situation might indicate an error. See Compiler Warning (level 1) CS4014.

The `async` keyword is contextual in that it's a keyword only when it modifies a method, a lambda expression, or an anonymous method. In all other contexts, it's interpreted as an identifier.

Example

The following example shows the structure and flow of control between an async event handler, `StartButton_Click`, and an async method, `ExampleMethodAsync`. The result from the async method is the number of characters of a web page. The code is suitable for a Windows Presentation Foundation (WPF) app or Windows Store app that you create in Visual Studio; see the code comments for setting up the app.

You can run this code in Visual Studio as a Windows Presentation Foundation (WPF) app or a Windows Store app. You need a Button control named `StartButton` and a Textbox control named `ResultsTextBox`. Remember to set the names and handler so that you have something like this:

XAML

```
<Button Content="Button" HorizontalAlignment="Left" Margin="88,77,0,0"
VerticalAlignment="Top" Width="75"
        Click="StartButton_Click" Name="StartButton"/>
<TextBox HorizontalAlignment="Left" Height="137" Margin="88,140,0,0"
TextWrapping="Wrap"
         Text="&lt;Enter a URL&gt;" VerticalAlignment="Top" Width="310"
Name="ResultsTextBox"/>
```

To run the code as a WPF app:

- Paste this code into the `MainWindow` class in MainWindow.xaml.cs.
- Add a reference to System.Net.Http.
- Add a `using` directive for System.Net.Http.

To run the code as a Windows Store app:

- Paste this code into the `MainPage` class in MainPage.xaml.cs.
- Add `using` directives for System.Net.Http and System.Threading.Tasks.

C#

```
private async void StartButton_Click(object sender, RoutedEventArgs e)
{
    // ExampleMethodAsync returns a Task<int>, which means that the method
    // eventually produces an int result. However, ExampleMethodAsync
returns
    // the Task<int> value as soon as it reaches an await.
    ResultsTextBox.Text += "\n";

    try
    {
        int length = await ExampleMethodAsync();
        // Note that you could put "await ExampleMethodAsync()" in the next
line where
```

```
        // "length" is, but due to when '+=' fetches the value of
ResultsTextBox, you
        // would not see the global side effect of ExampleMethodAsync
setting the text.
        ResultsTextBox.Text += String.Format("Length: {0:N0}\n", length);
    }
    catch (Exception)
    {
        // Process the exception if one occurs.
    }
}

public async Task<int> ExampleMethodAsync()
{
    var httpClient = new HttpClient();
    int exampleInt = (await
httpClient.GetStringAsync("http://msdn.microsoft.com")).Length;
    ResultsTextBox.Text += "Preparing to finish ExampleMethodAsync.\n";
    // After the following return statement, any method that's awaiting
    // ExampleMethodAsync (in this case, StartButton_Click) can get the
    // integer result.
    return exampleInt;
}
// The example displays the following output:
// Preparing to finish ExampleMethodAsync.
// Length: 53292
```

> **ⓘ Important**
>
> For more information about tasks and the code that executes while waiting for a task, see Asynchronous programming with async and await. For a full console example that uses similar elements, see Process asynchronous tasks as they complete (C#).

Return Types

An async method can have the following return types:

- Task
- Task<TResult>
- void. `async void` methods are generally discouraged for code other than event handlers because callers cannot `await` those methods and must implement a different mechanism to report successful completion or error conditions.
- Any type that has an accessible `GetAwaiter` method. The `System.Threading.Tasks.ValueTask<TResult>` type is one such implementation. It is available by adding the NuGet package `System.Threading.Tasks.Extensions`.

The async method can't declare any in, ref or out parameters, nor can it have a reference return value, but it can call methods that have such parameters.

You specify `Task<TResult>` as the return type of an async method if the return statement of the method specifies an operand of type `TResult`. You use `Task` if no meaningful value is returned when the method is completed. That is, a call to the method returns a `Task`, but when the `Task` is completed, any `await` expression that's awaiting the `Task` evaluates to `void`.

You use the `void` return type primarily to define event handlers, which require that return type. The caller of a `void`-returning async method can't await it and can't catch exceptions that the method throws.

You return another type, typically a value type, that has a `GetAwaiter` method to minimize memory allocations in performance-critical sections of code.

For more information and examples, see Async Return Types.

See also

- AsyncStateMachineAttribute
- await
- Asynchronous programming with async and await
- Process asynchronous tasks as they complete
- .NET blog: How async/await really works in C#

The const keyword

Article • 12/04/2024

You use the `const` keyword to declare a constant field or a local constant. Constant fields and locals aren't variables and can't be modified. Constants can be numbers, Boolean values, strings, or a null reference. Don't create a constant to represent information that you expect to change at any time. For example, don't use a constant field to store the price of a service, a product version number, or the brand name of a company. These values can change over time, and because compilers propagate constants, other code compiled with your libraries will have to be recompiled to see the changes. See also the readonly keyword. For example:

C#

```
const int X = 0;
public const double GravitationalConstant = 6.673e-11;
private const string ProductName = "Visual C#";
```

Interpolated strings can be constants, if all expressions used are also constant strings. This feature can improve the code that builds constant strings:

C#

```
const string Language = "C#";
const string Platform = ".NET";
const string FullProductName = $"{Platform} - Language: {Language}";
```

Remarks

The type of a constant declaration specifies the type of the members that the declaration introduces. The initializer of a local constant or a constant field must be a constant expression that can be implicitly converted to the target type.

A constant expression is an expression that can be fully evaluated at compile time. Therefore, the only possible values for constants of reference types are strings and a null reference.

The constant declaration can declare multiple constants, such as:

C#

```
public const double X = 1.0, Y = 2.0, Z = 3.0;
```

The `static` modifier isn't allowed in a constant declaration.

A constant can participate in a constant expression, as follows:

C#

```
public const int C1 = 5;
public const int C2 = C1 + 100;
```

> **Note**
>
> The readonly keyword differs from the `const` keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be initialized either at the declaration or in a constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, although a `const` field is a compile-time constant, the `readonly` field can be used for run-time constants, as in this line: `public static readonly uint l1 = (uint)DateTime.Now.Ticks;`

Examples

C#

```
public class ConstTest
{
    class SampleClass
    {
        public int x;
        public int y;
        public const int C1 = 5;
        public const int C2 = C1 + 5;

        public SampleClass(int p1, int p2)
        {
            x = p1;
            y = p2;
        }
    }

    static void Main()
    {
        var mC = new SampleClass(11, 22);
        Console.WriteLine($"x = {mC.x}, y = {mC.y}");
```

```
        Console.WriteLine($"C1 = {SampleClass.C1}, C2 = {SampleClass.C2}");
    }
}
/* Output
    x = 11, y = 22
    C1 = 5, C2 = 10
*/
```

The following example demonstrates how to declare a local constant:

C#

```
public class SealedTest
{
    static void Main()
    {
        const int C = 707;
        Console.WriteLine($"My local constant = {C}");
    }
}
// Output: My local constant = 707
```

C# language specification

For more information, see the following sections of the C# language specification:

- Constants
- Constant expressions

See also

- C# keywords
- readonly

The `event` keyword (C# reference)

Article • 03/15/2025

An ***event*** is a member that enables an object to trigger notifications. Event users can attach executable code for events by supplying ***event handlers***. The `event` keyword declares an ***event***. The event is of a delegate type. While an object triggers an event, the event invokes all supplied event handlers. Event handlers are delegate instances added to the event and executed when the event is raised. Event users can add or remove their event handlers on an event.

The following example shows how to declare and raise an event that uses EventHandler as the underlying delegate type. For the complete code example, see How to publish events that conform to .NET Guidelines. That sample demonstrates the generic EventHandler<TEventArgs> delegate type, how to subscribe to an event, and create an event handler method,

C#

```
public class SampleEventArgs
{
    public SampleEventArgs(string text) { Text = text; }
    public string Text { get; } // readonly
}

public class Publisher
{
    // Declare the delegate (if using non-generic pattern).
    public delegate void SampleEventHandler(object sender, SampleEventArgs
e);

    // Declare the event.
    public event SampleEventHandler SampleEvent;

    // Wrap the event in a protected virtual method
    // to enable derived classes to raise the event.
    protected virtual void RaiseSampleEvent()
    {
        // Raise the event in a thread-safe manner using the ?. operator.
        SampleEvent?.Invoke(this, new SampleEventArgs("Hello"));
    }
}
```

Events are multicast delegates that can only be invoked from within the class (or derived classes) or struct where they're declared (the publisher class). If other classes or structs subscribe to the event, their event handler methods are called when the publisher class raises the event. For more information and code examples, see Events and Delegates.

Events 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 event. For more information, see Access Modifiers.

Beginning with C# 14, events can be partial. Partial events have one defining declaration and one implementing declaration. The defining declaration must use the field-like syntax. The implementing declaration must declare the `add` and `remove` handlers.

Keywords and events

The following keywords apply to events.

Expand table

Keyword	Description	For more information
static	Makes the event available to callers at any time, even if no instance of the class exists.	Static Classes and Static Class Members
virtual	Allows derived classes to override the event behavior by using the override keyword.	Inheritance
sealed	Specifies that for derived classes it's no longer virtual.	
abstract	The compiler doesn't generate the `add` and `remove` event accessor blocks and therefore derived classes must provide their own implementation.	

An event can be declared as a static event by using the static keyword. Static events 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.

An event can be marked as a virtual event by using the virtual keyword. Derived classes can override the event behavior by using the override keyword. For more information, see Inheritance. An event overriding a virtual event can also be sealed, which specifies that for derived classes it's no longer virtual. Lastly, an event can be declared abstract, which means that the compiler doesn't generate the `add` and `remove` event accessor blocks. Therefore derived classes must provide their own implementation.

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

- C# Keywords
- add
- remove
- Modifiers
- How to combine delegates (Multicast Delegates)

extern (C# Reference)

Article • 03/12/2024

The `extern` modifier is used to declare a method that is implemented externally. A common use of the `extern` modifier is with the `DllImport` attribute when you are using Interop services to call into unmanaged code. In this case, the method must also be declared as `static`, as shown in the following example:

C#

```
[DllImport("avifil32.dll")]
private static extern void AVIFileInit();
```

The `extern` keyword can also define an external assembly alias, which makes it possible to reference different versions of the same component from within a single assembly. For more information, see extern alias.

It is an error to use the abstract and `extern` modifiers together to modify the same member. Using the `extern` modifier means that the method is implemented outside the C# code, whereas using the `abstract` modifier means that the method implementation is not provided in the class.

The extern keyword has more limited uses in C# than in C++. To compare the C# keyword with the C++ keyword, see Using extern to Specify Linkage in the C++ Language Reference.

Example 1

In this example, the program receives a string from the user and displays it inside a message box. The program uses the `MessageBox` method imported from the User32.dll library.

C#

```
//using System.Runtime.InteropServices;
class ExternTest
{
    [DllImport("User32.dll", CharSet=CharSet.Unicode)]
    public static extern int MessageBox(IntPtr h, string m, string c, int
type);

    static int Main()
    {
```

```
        string myString;
        Console.Write("Enter your message: ");
        myString = Console.ReadLine();
        return MessageBox((IntPtr)0, myString, "My Message Box", 0);
    }
}
```

Example 2

This example illustrates a C# program that calls into a C library (a native DLL).

1. Create the following C file and name it `cmdll.c`:

```
C

// cmdll.c
// Compile with: -LD
int __declspec(dllexport) SampleMethod(int i)
{
  return i*10;
}
```

2. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cmdll.c` file by typing **cl -LD cmdll.c** at the command prompt.

3. In the same directory, create the following C# file and name it `cm.cs`:

```
C#

// cm.cs
using System;
using System.Runtime.InteropServices;
public class MainClass
{
    [DllImport("Cmdll.dll")]
      public static extern int SampleMethod(int x);

    static void Main()
    {
        Console.WriteLine("SampleMethod() returns {0}.",
SampleMethod(5));
    }
}
```

4. Open a Visual Studio x64 (or x32) Native Tools Command Prompt window from the Visual Studio installation directory and compile the `cm.cs` file by typing:

> **csc cm.cs** (for the x64 command prompt) —or— **csc -platform:x86 cm.cs** (for the x32 command prompt)

This will create the executable file `cm.exe`.

5. Run `cm.exe`. The `SampleMethod` method passes the value 5 to the DLL file, which returns the value multiplied by 10. The program produces the following output:

```
Output

SampleMethod() returns 50.
```

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

- System.Runtime.InteropServices.DllImportAttribute
- C# Keywords
- Modifiers

in (Generic Modifier) (C# Reference)

Article • 02/25/2022

For generic type parameters, the `in` keyword specifies that the type parameter is contravariant. You can use the `in` keyword in generic interfaces and delegates.

Contravariance enables you to use a less derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement contravariant interfaces and implicit conversion of delegate types. Covariance and contravariance in generic type parameters are supported for reference types, but they are not supported for value types.

A type can be declared contravariant in a generic interface or delegate only if it defines the type of a method's parameters and not of a method's return type. `In`, `ref`, and `out` parameters must be invariant, meaning they are neither covariant nor contravariant.

An interface that has a contravariant type parameter allows its methods to accept arguments of less derived types than those specified by the interface type parameter. For example, in the IComparer<T> interface, type T is contravariant, you can assign an object of the `IComparer<Person>` type to an object of the `IComparer<Employee>` type without using any special conversion methods if `Employee` inherits `Person`.

A contravariant delegate can be assigned another delegate of the same type, but with a less derived generic type parameter.

For more information, see Covariance and Contravariance.

Contravariant generic interface

The following example shows how to declare, extend, and implement a contravariant generic interface. It also shows how you can use implicit conversion for classes that implement this interface.

C#

```
// Contravariant interface.
interface IContravariant<in A> { }

// Extending contravariant interface.
interface IExtContravariant<in A> : IContravariant<A> { }

// Implementing contravariant interface.
class Sample<A> : IContravariant<A> { }
```

```
class Program
{
    static void Test()
    {
        IContravariant<Object> iobj = new Sample<Object>();
        IContravariant<String> istr = new Sample<String>();

        // You can assign iobj to istr because
        // the IContravariant interface is contravariant.
        istr = iobj;
    }
}
```

Contravariant generic delegate

The following example shows how to declare, instantiate, and invoke a contravariant generic delegate. It also shows how you can implicitly convert a delegate type.

C#

```
// Contravariant delegate.
public delegate void DContravariant<in A>(A argument);

// Methods that match the delegate signature.
public static void SampleControl(Control control)
{ }
public static void SampleButton(Button button)
{ }

public void Test()
{

    // Instantiating the delegates with the methods.
    DContravariant<Control> dControl = SampleControl;
    DContravariant<Button> dButton = SampleButton;

    // You can assign dControl to dButton
    // because the DContravariant delegate is contravariant.
    dButton = dControl;

    // Invoke the delegate.
    dButton(new Button());
}
```

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

- out
- Covariance and Contravariance
- Modifiers

new modifier (C# Reference)

Article • 04/12/2023

When used as a declaration modifier, the `new` keyword explicitly hides a member that is inherited from a base class. When you hide an inherited member, the derived version of the member replaces the base class version. This assumes that the base class version of the member is visible, as it would already be hidden if it were marked as `private` or, in some cases, `internal`. Although you can hide `public` or `protected` members without using the `new` modifier, you get a compiler warning. If you use `new` to explicitly hide a member, it suppresses this warning.

You can also use the `new` keyword to create an instance of a type or as a generic type constraint.

To hide an inherited member, declare it in the derived class by using the same member name, and modify it with the `new` keyword. For example:

C#

```
public class BaseC
{
    public int x;
    public void Invoke() { }
}
public class DerivedC : BaseC
{
    new public void Invoke() { }
}
```

In this example, `BaseC.Invoke` is hidden by `DerivedC.Invoke`. The field `x` is not affected because it is not hidden by a similar name.

Name hiding through inheritance takes one of the following forms:

- Generally, a constant, field, property, or type that is introduced in a class or struct hides all base class members that share its name. There are special cases. For example, if you declare a new field with name `N` to have a type that is not invocable, and a base type declares `N` to be a method, the new field does not hide the base declaration in invocation syntax. For more information, see the Member lookup section of the C# language specification.
- A method introduced in a class or struct hides properties, fields, and types that share that name in the base class. It also hides all base class methods that have the

same signature.

- An indexer introduced in a class or struct hides all base class indexers that have the same signature.

It is an error to use both `new` and override on the same member, because the two modifiers have mutually exclusive meanings. The `new` modifier creates a new member with the same name and causes the original member to become hidden. The `override` modifier extends the implementation for an inherited member.

Using the `new` modifier in a declaration that does not hide an inherited member generates a warning.

Examples

In this example, a base class, `BaseC`, and a derived class, `DerivedC`, use the same field name `x`, which hides the value of the inherited field. The example demonstrates the use of the `new` modifier. It also demonstrates how to access the hidden members of the base class by using their fully qualified names.

C#

```
public class BaseC
{
    public static int x = 55;
    public static int y = 22;
}

public class DerivedC : BaseC
{
    // Hide field 'x'.
    new public static int x = 100;

    static void Main()
    {
        // Display the new value of x:
        Console.WriteLine(x);

        // Display the hidden value of x:
        Console.WriteLine(BaseC.x);

        // Display the unhidden member y:
        Console.WriteLine(y);
    }
}
/*
Output:
100
```

```
55
22
*/
```

In this example, a nested class hides a class that has the same name in the base class. The example demonstrates how to use the `new` modifier to eliminate the warning message and how to access the hidden class members by using their fully qualified names.

C#

```
public class BaseC
{
    public class NestedC
    {
        public int x = 200;
        public int y;
    }
}

public class DerivedC : BaseC
{
    // Nested type hiding the base type members.
    new public class NestedC
    {
        public int x = 100;
        public int y;
        public int z;
    }

    static void Main()
    {
        // Creating an object from the overlapping class:
        NestedC c1  = new NestedC();

        // Creating an object from the hidden class:
        BaseC.NestedC c2 = new BaseC.NestedC();

        Console.WriteLine(c1.x);
        Console.WriteLine(c2.x);
    }
}
/*
Output:
100
200
*/
```

If you remove the `new` modifier, the program will still compile and run, but you will get the following warning:

```text
The keyword new is required on 'MyDerivedC.x' because it hides inherited
member 'MyBaseC.x'.
```

C# language specification

For more information, see The new modifier section of the C# language specification.

See also

- C# Keywords
- Modifiers
- Versioning with the Override and New Keywords
- Knowing When to Use Override and New Keywords

out (generic modifier) (C# Reference)

Article • 09/15/2021

For generic type parameters, the `out` keyword specifies that the type parameter is covariant. You can use the `out` keyword in generic interfaces and delegates.

Covariance enables you to use a more derived type than that specified by the generic parameter. This allows for implicit conversion of classes that implement covariant interfaces and implicit conversion of delegate types. Covariance and contravariance are supported for reference types, but they are not supported for value types.

An interface that has a covariant type parameter enables its methods to return more derived types than those specified by the type parameter. For example, because in .NET Framework 4, in IEnumerable<T>, type T is covariant, you can assign an object of the `IEnumerable(Of String)` type to an object of the `IEnumerable(Of Object)` type without using any special conversion methods.

A covariant delegate can be assigned another delegate of the same type, but with a more derived generic type parameter.

For more information, see Covariance and Contravariance.

Example - covariant generic interface

The following example shows how to declare, extend, and implement a covariant generic interface. It also shows how to use implicit conversion for classes that implement a covariant interface.

C#

```
// Covariant interface.
interface ICovariant<out R> { }

// Extending covariant interface.
interface IExtCovariant<out R> : ICovariant<R> { }

// Implementing covariant interface.
class Sample<R> : ICovariant<R> { }

class Program
{
    static void Test()
    {
        ICovariant<Object> iobj = new Sample<Object>();
        ICovariant<String> istr = new Sample<String>();
```

```
        // You can assign istr to iobj because
        // the ICovariant interface is covariant.
        iobj = istr;
    }
}
```

In a generic interface, a type parameter can be declared covariant if it satisfies the following conditions:

- The type parameter is used only as a return type of interface methods and not used as a type of method arguments.

  > **Note**
  >
  > There is one exception to this rule. If in a covariant interface you have a contravariant generic delegate as a method parameter, you can use the covariant type as a generic type parameter for this delegate. For more information about covariant and contravariant generic delegates, see Variance in Delegates and Using Variance for Func and Action Generic Delegates.

- The type parameter is not used as a generic constraint for the interface methods.

Example - covariant generic delegate

The following example shows how to declare, instantiate, and invoke a covariant generic delegate. It also shows how to implicitly convert delegate types.

C#

```
// Covariant delegate.
public delegate R DCovariant<out R>();

// Methods that match the delegate signature.
public static Control SampleControl()
{ return new Control(); }

public static Button SampleButton()
{ return new Button(); }

public void Test()
{
    // Instantiate the delegates with the methods.
    DCovariant<Control> dControl = SampleControl;
    DCovariant<Button> dButton = SampleButton;
```

```
    // You can assign dButton to dControl
    // because the DCovariant delegate is covariant.
    dControl = dButton;

    // Invoke the delegate.
    dControl();
}
```

In a generic delegate, a type can be declared covariant if it is used only as a method return type and not used for method arguments.

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

- Variance in Generic Interfaces
- in
- Modifiers

override (C# reference)

Article • 03/30/2024

The `override` modifier is required to extend or modify the abstract or virtual implementation of an inherited method, property, indexer, or event.

In the following example, the `Square` class must provide an overridden implementation of `GetArea` because `GetArea` is inherited from the abstract `Shape` class:

C#

```
abstract class Shape
{
    public abstract int GetArea();
}

class Square : Shape
{
    private int _side;

    public Square(int n) => _side = n;

    // GetArea method is required to avoid a compile-time error.
    public override int GetArea() => _side * _side;

    static void Main()
    {
        var sq = new Square(12);
        Console.WriteLine($"Area of the square = {sq.GetArea()}");
    }
}
// Output: Area of the square = 144
```

An `override` method provides a new implementation of the method inherited from a base class. The method that is overridden by an `override` declaration is known as the overridden base method. An `override` method must have the same signature as the overridden base method. `override` methods support covariant return types. In particular, the return type of an `override` method can derive from the return type of the corresponding base method.

You cannot override a non-virtual or static method. The overridden base method must be `virtual`, `abstract`, or `override`.

An `override` declaration cannot change the accessibility of the `virtual` method. Both the `override` method and the `virtual` method must have the same access level

modifier.

You cannot use the `new`, `static`, or `virtual` modifiers to modify an `override` method.

An overriding property declaration must specify exactly the same access modifier, type, and name as the inherited property. Read-only overriding properties support covariant return types. The overridden property must be `virtual`, `abstract`, or `override`.

For more information about how to use the `override` keyword, see Versioning with the Override and New Keywords and Knowing when to use Override and New Keywords. For information about inheritance, see Inheritance.

Example

This example defines a base class named `Employee`, and a derived class named `SalesEmployee`. The `SalesEmployee` class includes an extra field, `salesbonus`, and overrides the method `CalculatePay` in order to take it into account.

C#

```
class TestOverride
{
    public class Employee
    {
        public string Name { get; }

        // Basepay is defined as protected, so that it may be
        // accessed only by this class and derived classes.
        protected decimal _basepay;

        // Constructor to set the name and basepay values.
        public Employee(string name, decimal basepay)
        {
            Name = name;
            _basepay = basepay;
        }

        // Declared virtual so it can be overridden.
        public virtual decimal CalculatePay()
        {
            return _basepay;
        }
    }

    // Derive a new class from Employee.
    public class SalesEmployee : Employee
    {
        // New field that will affect the base pay.
        private decimal _salesbonus;
```

```
        // The constructor calls the base-class version, and
        // initializes the salesbonus field.
        public SalesEmployee(string name, decimal basepay, decimal
salesbonus)
            : base(name, basepay)
        {
            _salesbonus = salesbonus;
        }

        // Override the CalculatePay method
        // to take bonus into account.
        public override decimal CalculatePay()
        {
            return _basepay + _salesbonus;
        }
    }

    static void Main()
    {
        // Create some new employees.
        var employee1 = new SalesEmployee("Alice", 1000, 500);
        var employee2 = new Employee("Bob", 1200);

        Console.WriteLine($"Employee1 {employee1.Name} earned:
{employee1.CalculatePay()}");
        Console.WriteLine($"Employee2 {employee2.Name} earned:
{employee2.CalculatePay()}");
    }
}
/*
    Output:
    Employee1 Alice earned: 1500
    Employee2 Bob earned: 1200
*/
```

C# language specification

For more information, see the Override methods section of the C# language specification.

For more information about covariant return types, see the feature proposal note.

See also

- Inheritance
- C# keywords
- Modifiers
- abstract

- virtual
- new (modifier)
- Polymorphism

readonly (C# Reference)

The `readonly` keyword is a modifier that can be used in five contexts:

- In a field declaration, `readonly` indicates that assignment to the field can only occur as part of the declaration or in a constructor in the same class. A readonly field can be assigned and reassigned multiple times within the field declaration and constructor.

  A `readonly` field can't be assigned after the constructor exits. This rule has different implications for value types and reference types:
  - Because value types directly contain their data, a field that is a `readonly` value type is immutable.
  - Because reference types contain a reference to their data, a field that is a `readonly` reference type must always refer to the same object. That object might not be immutable. The `readonly` modifier prevents replacing the field value with a different instance of the reference type. However, the modifier doesn't prevent the instance data of the field from being modified through the read-only field.

  > ⚠ **Warning**
  >
  > An externally visible type that contains an externally visible read-only field that is a mutable reference type may be a security vulnerability and may trigger warning CA2104 : "Do not declare read only mutable reference types."

- In a `readonly struct` type definition, `readonly` indicates that the structure type is immutable. For more information, see the readonly struct section of the Structure types article.

- In an instance member declaration within a structure type, `readonly` indicates that an instance member doesn't modify the state of the structure. For more information, see the readonly instance members section of the Structure types article.

- In a ref readonly method return, the `readonly` modifier indicates that method returns a reference and writes aren't allowed to that reference.
  - To declare a ref readonly parameter to a method.

Readonly field example

In this example, the value of the field `year` can't be changed in the method `ChangeYear`, even though it was assigned a value in the class constructor:

C#

```
class Age
{
    private readonly int _year;
    Age(int year)
    {
        _year = year;
    }
    void ChangeYear()
    {
        //_year = 1967; // Compile error if uncommented.
    }
}
```

You can assign a value to a `readonly` field only in the following contexts:

- When the variable is initialized in the declaration, for example:

  C#

  ```
  public readonly int y = 5;
  ```

- In an instance constructor of the class that contains the instance field declaration.
- In the static constructor of the class that contains the static field declaration.

These constructor contexts are also the only contexts in which it's valid to pass a `readonly` field as an out or ref parameter.

> **Note**
>
> The `readonly` keyword is different from the const keyword. A `const` field can only be initialized at the declaration of the field. A `readonly` field can be assigned multiple times in the field declaration and in any constructor. Therefore, `readonly` fields can have different values depending on the constructor used. Also, while a `const` field is a compile-time constant, the `readonly` field can be used for run-time constants as in the following example:
>
> C#
>
> ```
> public static readonly uint timeStamp = (uint)DateTime.Now.Ticks;
> ```

C#

```
public class SamplePoint
{
    public int x;
    // Initialize a readonly field
    public readonly int y = 25;
    public readonly int z;

    public SamplePoint()
    {
        // Initialize a readonly instance field
        z = 24;
    }

    public SamplePoint(int p1, int p2, int p3)
    {
        x = p1;
        y = p2;
        z = p3;
    }

    public static void Main()
    {
        SamplePoint p1 = new SamplePoint(11, 21, 32);   // OK
        Console.WriteLine($"p1: x={p1.x}, y={p1.y}, z={p1.z}");
        SamplePoint p2 = new SamplePoint();
        p2.x = 55;   // OK
        Console.WriteLine($"p2: x={p2.x}, y={p2.y}, z={p2.z}");
    }
    /*
     Output:
        p1: x=11, y=21, z=32
        p2: x=55, y=25, z=24
    */
}
```

In the preceding example, if you use a statement like the following example:

C#

```
p2.y = 66;        // Error
```

you get the compiler error message:

A readonly field cannot be assigned to (except in a constructor or a variable initializer)

Readonly instance members

You can also use the `readonly` modifier to declare that an instance member doesn't modify the state of a struct.

C#

```
public readonly double Sum()
{
    return X + Y;
}
```

> **Note**
>
> In the case of a read/write property, you can add the `readonly` modifier to the `get` accessor. Some `get` accessors may perform a calculation and cache the result, rather than simply returning the value of a private field. Adding the `readonly` modifier to the `get` accessor guarantees that the `get` accessor doesn't modify the internal state of the object by caching any result.

You can find more examples in the readonly instance members section of the Structure types article.

Ref readonly return example

The `readonly` modifier on a `ref return` indicates that the returned reference can't be modified. The following example returns a reference to the origin. It uses the `readonly` modifier to indicate that callers can't modify the origin:

C#

```
private static readonly SamplePoint s_origin = new SamplePoint(0, 0, 0);
public static ref readonly SamplePoint Origin => ref s_origin;
```

The type returned doesn't need to be a `readonly struct`. Any type that can be returned by `ref` can be returned by `ref readonly`.

Readonly ref readonly return example

A `ref readonly return` can also be used with `readonly` instance members on `struct` types:

C#

```
public struct ReadonlyRefReadonlyExample
{
    private int _data;

    public readonly ref readonly int ReadonlyRefReadonly(ref int reference)
```

```
    {
        // _data = 1; // Compile error if uncommented.
        return ref reference;
    }
}
```

The method essentially returns a `readonly` reference together with the instance member (in this case a method) being `readonly` (not able to modify any instance fields).

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

- Add readonly modifier (style rule IDE0044)
- C# Keywords
- Modifiers
- const
- Fields

Last updated on 11/01/2025

sealed (C# Reference)

Article • 01/11/2025

When applied to a class, the `sealed` modifier prevents other classes from inheriting from it. In the following example, class `B` inherits from class `A`, but no class can inherit from class `B`.

C#

```
class A {}
sealed class B : A {}
```

You can also use the `sealed` modifier on a method or property that overrides a virtual method or property in a base class. This enables you to allow classes to derive from your class and prevent them from overriding specific virtual methods or properties.

Example

In the following example, `Z` inherits from `Y` but `Z` cannot override the virtual function `F` that is declared in `X` and sealed in `Y`.

C#

```
class X
{
    protected virtual void F() { Console.WriteLine("X.F"); }
    protected virtual void F2() { Console.WriteLine("X.F2"); }
}

class Y : X
{
    sealed protected override void F() { Console.WriteLine("Y.F"); }
    protected override void F2() { Console.WriteLine("Y.F2"); }
}

class Z : Y
{
    // Attempting to override F causes compiler error CS0239.
    // protected override void F() { Console.WriteLine("Z.F"); }

    // Overriding F2 is allowed.
    protected override void F2() { Console.WriteLine("Z.F2"); }
}
```

When you define new methods or properties in a class, you can prevent deriving classes from overriding them by not declaring them as virtual. When you override a `virtual` member declared in a base type, you can prevent deriving types from overriding them by using sealed keyword as in the following example:

```
public sealed override string ToString() => Value;
```

It is an error to use the abstract modifier with a sealed class, because an abstract class must be inherited by a class that provides an implementation of the abstract methods or properties.

When applied to a method or property, the `sealed` modifier must always be used with override.

Because structs are implicitly sealed, they cannot be inherited.

For more information, see Inheritance.

For more examples, see Abstract and Sealed Classes and Class Members.

C#

```
sealed class SealedClass
{
    public int x;
    public int y;
}

class SealedTest2
{
    static void Main()
    {
        var sc = new SealedClass();
        sc.x = 110;
        sc.y = 150;
        Console.WriteLine($"x = {sc.x}, y = {sc.y}");
    }
}
// Output: x = 110, y = 150
```

In the previous example, you might try to inherit from the sealed class by using the following statement:

```
class MyDerivedC: SealedClass {} // Error
```

The result is an error message:

```
'MyDerivedC': cannot derive from sealed type 'SealedClass'
```

Remarks

To determine whether to seal a class, method, or property, you should generally consider the following two points:

- The potential benefits that deriving classes might gain through the ability to customize your class.
- The potential that deriving classes could modify your classes in such a way that they would no longer work correctly or as expected.

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

- C# Keywords
- Static Classes and Static Class Members
- Abstract and Sealed Classes and Class Members
- Access Modifiers
- Modifiers
- override
- virtual

static (C# Reference)

07/09/2025

This page covers the `static` modifier keyword. The `static` keyword is also part of the using static directive.

Use the `static` modifier to declare a static member, which belongs to the type itself rather than to a specific object. The `static` modifier can be used to declare `static` classes. In classes, interfaces, and structs, you may add the `static` modifier to fields, methods, properties, operators, events, and constructors. The `static` modifier can't be used with indexers or finalizers. For more information, see Static Classes and Static Class Members.

You can add the `static` modifier to a local function. A static local function can't capture local variables or instance state.

C#

```
class Calc1
{
    public void CalculateSum()
    {
        int a = 3;
        int b = 7;

        // Static local function - cannot access 'a' or 'b' directly
        static int Add(int x, int y)
        {
            return x + y;
        }

        int result = Add(a, b);
        Console.WriteLine($"Sum: {result}");
    }
}
  /*
 Output:
 Sum: 10
 */
```

You can add the `static` modifier to a lambda expression or anonymous method. A static lambda or anonymous method can't capture local variables or instance state.

C#

```
class Calc2
{
    static void Main()
```

```
    {
        Func<int, int, int> add = static (a, b) => a + b;

        int result = add(5, 10);
        Console.WriteLine($"Sum: {result}");
    }
}
/*
Output:
Sum: 15
*/
```

Example - static class

The following class is declared as `static` and contains only `static` methods:

C#

```
static class CompanyEmployee
{
    public static void DoSomething() { /*...*/ }
    public static void DoSomethingElse() { /*...*/  }
}
```

A constant or type declaration is implicitly a `static` member. A `static` member can't be referenced through an instance. Instead, it's referenced through the type name. For example, consider the following class:

C#

```
public class MyBaseC
{
    public struct MyStruct
    {
        public static int x = 100;
    }
}
```

To refer to the `static` member `x`, use the fully qualified name, `MyBaseC.MyStruct.x`, unless the member is accessible from the same scope:

C#

```
Console.WriteLine(MyBaseC.MyStruct.x);
```

While an instance of a class contains a separate copy of all instance fields of the class, there's only one copy of each `static` field. For generic types, each closed generic type has its own copy of static fields. Static fields marked with ThreadStaticAttribute have one copy per thread.

It isn't possible to use this to reference `static` methods or property accessors.

If the `static` keyword is applied to a class, all the members of the class must be `static`.

Classes, interfaces, and `static` classes may have `static` constructors. A `static` constructor is called at some point between when the program starts and the class is instantiated.

> **Note**
>
> The `static` keyword has more limited uses than in C++. To compare with the C++ keyword, see Storage classes (C++).

To demonstrate `static` members, consider a class that represents a company employee. Assume that the class contains a method to count employees and a field to store the number of employees. Both the method and the field don't belong to any one employee instance. Instead, they belong to the class of employees as a whole. They should be declared as `static` members of the class.

Example - static field and method

This example reads the name and ID of a new employee, increments the employee counter by one, and displays the information for the new employee and the new number of employees. This program reads the current number of employees from the keyboard.

C#

```
public class Employee4
{
    public string id;
    public string name;

    public Employee4()
    {
    }

    public Employee4(string name, string id)
    {
        this.name = name;
        this.id = id;
    }
```

```
    public static int employeeCounter;

    public static int AddEmployee()
    {
        return ++employeeCounter;
    }
}

class MainClass : Employee4
{
    static void Main()
    {
        Console.Write("Enter the employee's name: ");
        string name = Console.ReadLine();
        Console.Write("Enter the employee's ID: ");
        string id = Console.ReadLine();

        // Create and configure the employee object.
        Employee4 e = new Employee4(name, id);
        Console.Write("Enter the current number of employees: ");
        string n = Console.ReadLine();
        Employee4.employeeCounter = Int32.Parse(n);
        Employee4.AddEmployee();

        // Display the new information.
        Console.WriteLine($"Name: {e.name}");
        Console.WriteLine($"ID:   {e.id}");
        Console.WriteLine($"New Number of Employees:
{Employee4.employeeCounter}");
    }
}
/*
Input:
Matthias Berndt
AF643G
15
 *
Sample Output:
Enter the employee's name: Matthias Berndt
Enter the employee's ID: AF643G
Enter the current number of employees: 15
Name: Matthias Berndt
ID:   AF643G
New Number of Employees: 16
*/
```

Example - static initialization

This example shows that you can initialize a `static` field by using another `static` field that is not yet declared. The results will be undefined until you explicitly assign a value to the `static` field.

C#

```
class Test
{
    static int x = y;
    static int y = 5;

    static void Main()
    {
        Console.WriteLine(Test.x);
        Console.WriteLine(Test.y);

        Test.x = 99;
        Console.WriteLine(Test.x);
    }
}
/*
Output:
    0
    5
    99
*/
```

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

- C# Keywords
- Modifiers
- using static directive
- Static Classes and Static Class Members

unsafe (C# Reference)

Article • 03/30/2024

The `unsafe` keyword denotes an unsafe context, which is required for any operation involving pointers. For more information, see Unsafe Code and Pointers.

You can use the `unsafe` modifier in the declaration of a type or a member. The entire textual extent of the type or member is therefore considered an unsafe context. For example, the following is a method declared with the `unsafe` modifier:

```
C#

unsafe static void FastCopy(byte[] src, byte[] dst, int count)
{
    // Unsafe context: can use pointers here.
}
```

The scope of the unsafe context extends from the parameter list to the end of the method, so pointers can also be used in the parameter list:

```
C#

unsafe static void FastCopy ( byte* ps, byte* pd, int count ) {...}
```

You can also use an unsafe block to enable the use of an unsafe code inside this block. For example:

```
C#

unsafe
{
    // Unsafe context: can use pointers here.
}
```

To compile unsafe code, you must specify the **AllowUnsafeBlocks** compiler option. Unsafe code is not verifiable by the common language runtime.

Example

```
C#

// compile with: -unsafe
class UnsafeTest
```

```
{
    // Unsafe method: takes pointer to int.
    unsafe static void SquarePtrParam(int* p)
    {
        *p *= *p;
    }

    unsafe static void Main()
    {
        int i = 5;
        // Unsafe method: uses address-of operator (&).
        SquarePtrParam(&i);
        Console.WriteLine(i);
    }
}
// Output: 25
```

C# language specification

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

See also

- C# keywords
- fixed statement
- Unsafe code, pointer types, and function pointers
- Pointer related operators

virtual (C# Reference)

Article • 10/08/2024

The `virtual` keyword is used to modify a method, property, indexer, or event declaration and allow for it to be overridden in a derived class. For example, this method can be overridden by any class that inherits it:

C#

```
public virtual double Area()
{
    return x * y;
}
```

The implementation of a virtual member can be changed by an overriding member in a derived class. For more information about how to use the `virtual` keyword, see Versioning with the Override and New Keywords and Knowing When to Use Override and New Keywords.

Remarks

When a virtual method is invoked, the run-time type of the object is checked for an overriding member. The overriding member in the most derived class is called, which might be the original member, if no derived class has overridden the member.

By default, methods are non-virtual. You cannot override a non-virtual method.

The following example shows a virtual property:

C#

```
class MyBaseClass
{
    // virtual automatically implemented property. Overrides can only
    // provide specialized behavior if they implement get and set accessors.
    public virtual string Name { get; set; }

    // ordinary virtual property with backing field
    private int _num;
    public virtual int Number
    {
        get { return _num; }
        set { _num = value; }
    }
}
```

```
class MyDerivedClass : MyBaseClass
{
    private string _name;

    // Override automatically implemented property with ordinary property
    // to provide specialized accessor behavior.
    public override string Name
    {
        get
        {
            return _name;
        }
        set
        {
            if (!string.IsNullOrEmpty(value))
            {
                _name = value;
            }
            else
            {
                _name = "Unknown";
            }
        }
    }
}
```

Virtual properties behave like virtual methods, except for the differences in declaration and invocation syntax.

- A virtual inherited property can be overridden in a derived class by including a property declaration that uses the `override` modifier.

Example

In this example, the `Shape` class contains the two coordinates `x`, `y`, and the `Area()` virtual method. Different shape classes such as `Circle`, `Cylinder`, and `Sphere` inherit the `Shape` class, and the surface area is calculated for each figure. Each derived class has its own override implementation of `Area()`.

Notice that the inherited classes `Circle`, `Cylinder`, and `Sphere` all use constructors that initialize the base class, as shown in the following declaration.

C#

```
public Cylinder(double r, double h): base(r, h) {}
```

The following program calculates and displays the appropriate area for each figure by invoking the appropriate implementation of the `Area()` method, according to the object that is associated with the method.

C#

```
class TestClass
{
    public class Shape
    {
        public const double PI = Math.PI;
        protected double _x, _y;

        public Shape()
        {
        }

        public Shape(double x, double y)
        {
            _x = x;
            _y = y;
        }

        public virtual double Area()
        {
            return _x * _y;
        }
    }

    public class Circle : Shape
    {
        public Circle(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return PI * _x * _x;
        }
    }

    public class Sphere : Shape
    {
        public Sphere(double r) : base(r, 0)
        {
        }

        public override double Area()
        {
            return 4 * PI * _x * _x;
        }
    }
```

```
    public class Cylinder : Shape
    {
        public Cylinder(double r, double h) : base(r, h)
        {
        }

        public override double Area()
        {
            return 2 * PI * _x * _x + 2 * PI * _x * _y;
        }
    }

    static void Main()
    {
        double r = 3.0, h = 5.0;
        Shape c = new Circle(r);
        Shape s = new Sphere(r);
        Shape l = new Cylinder(r, h);
        // Display results.
        Console.WriteLine($"Area of Circle   = {c.Area():F2}");
        Console.WriteLine($"Area of Sphere   = {s.Area():F2}");
        Console.WriteLine($"Area of Cylinder = {l.Area():F2}");
    }
}
/*
Output:
Area of Circle   = 28.27
Area of Sphere   = 113.10
Area of Cylinder = 150.80
*/
```

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

- Polymorphism
- abstract
- override
- new (modifier)

volatile (C# Reference)

07/16/2025

The `volatile` keyword indicates that a field might be modified by multiple threads that are executing at the same time. The compiler, the runtime system, and even hardware may rearrange reads and writes to memory locations for performance reasons. Fields that are declared `volatile` are excluded from certain kinds of optimizations. There is no guarantee of a single total ordering of volatile writes as seen from all threads of execution. For more information, see the Volatile class.

> **⊗ Caution**
>
> The `volatile` keyword is often misunderstood and misused in multithreaded programming. In most scenarios, you should use safer and more reliable alternatives instead of `volatile`. Modern .NET provides better concurrency tools like the **Interlocked** class, the **lock** statement, or higher-level synchronization primitives. These alternatives provide clearer semantics and stronger guarantees than `volatile`. Consider using `volatile` only in rare, advanced scenarios where you fully understand its limitations and have verified it's the appropriate solution.

> **ⓘ Note**
>
> On a multiprocessor system, a volatile read operation does not guarantee to obtain the latest value written to that memory location by any processor. Similarly, a volatile write operation does not guarantee that the value written would be immediately visible to other processors.

The `volatile` keyword can be applied to fields of these types:

- Reference types.
- Pointer types (in an unsafe context). Note that although the pointer itself can be volatile, the object that it points to cannot. In other words, you cannot declare a "pointer to volatile."
- Simple types such as `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, and `bool`.
- An `enum` type with one of the following base types: `byte`, `sbyte`, `short`, `ushort`, `int`, or `uint`.
- Generic type parameters known to be reference types.
- IntPtr and UIntPtr.

Other types, including `double` and `long`, cannot be marked `volatile` because reads and writes to fields of those types cannot be guaranteed to be atomic. To protect multi-threaded access to those types of fields, use the Interlocked class members or protect access using the lock statement.

For most multithreaded scenarios, even with supported types, prefer using Interlocked operations, lock statements, or other synchronization primitives instead of `volatile`. These alternatives are less prone to subtle concurrency bugs.

The `volatile` keyword can only be applied to fields of a `class` or `struct`. Local variables cannot be declared `volatile`.

Alternatives to volatile

In most cases, you should use one of these safer alternatives instead of `volatile`:

- **Interlocked operations**: Provide atomic operations for numeric types and reference assignments. These are generally faster and provide stronger guarantees than `volatile`.
- **lock statement**: Provides mutual exclusion and memory barriers. Use for protecting larger critical sections.
- **Volatile class**: Provides explicit volatile read and write operations with clearer semantics than the `volatile` keyword.
- **Higher-level synchronization primitives**: Such as ReaderWriterLockSlim, Semaphore, or concurrent collections from System.Collections.Concurrent.

The `volatile` keyword doesn't provide atomicity for operations other than assignment, doesn't prevent race conditions, and doesn't provide ordering guarantees for other memory operations. These limitations make it unsuitable for most concurrency scenarios.

Example

The following example shows how to declare a public field variable as `volatile`.

C#

```
class VolatileTest
{
    public volatile int sharedStorage;

    public void Test(int i)
    {
        sharedStorage = i;
```

```
    }
}
```

The following example demonstrates how an auxiliary or worker thread can be created and used to perform processing in parallel with that of the primary thread. For more information about multithreading, see Managed Threading.

C#

```
public class Worker
{
    // This method is called when the thread is started.
    public void DoWork()
    {
        bool work = false;
        while (!_shouldStop)
        {
            work = !work; // simulate some work
        }
        Console.WriteLine("Worker thread: terminating gracefully.");
    }
    public void RequestStop()
    {
        _shouldStop = true;
    }
    // Keyword volatile is used as a hint to the compiler that this data
    // member is accessed by multiple threads.
    private volatile bool _shouldStop;
}

public class WorkerThreadExample
{
    public static void Main()
    {
        // Create the worker thread object. This does not start the thread.
        Worker workerObject = new Worker();
        Thread workerThread = new Thread(workerObject.DoWork);

        // Start the worker thread.
        workerThread.Start();
        Console.WriteLine("Main thread: starting worker thread...");

        // Loop until the worker thread activates.
        while (!workerThread.IsAlive)
            ;

        // Put the main thread to sleep for 500 milliseconds to
        // allow the worker thread to do some work.
        Thread.Sleep(500);

        // Request that the worker thread stop itself.
        workerObject.RequestStop();
```

```
        // Use the Thread.Join method to block the current thread
        // until the object's thread terminates.
        workerThread.Join();
        Console.WriteLine("Main thread: worker thread has terminated.");
    }
    // Sample output:
    // Main thread: starting worker thread...
    // Worker thread: terminating gracefully.
    // Main thread: worker thread has terminated.
}
```

With the `volatile` modifier added to the declaration of `_shouldStop` in place, you'll always get the same results (similar to the excerpt shown in the preceding code). However, without that modifier on the `_shouldStop` member, the behavior is unpredictable. The `DoWork` method may optimize the member access, resulting in reading stale data. Because of the nature of multi-threaded programming, the number of stale reads is unpredictable. Different runs of the program will produce somewhat different results.

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

- C# language specification: volatile keyword
- C# Keywords
- Modifiers
- lock statement
- Interlocked
- Volatile
- System.Collections.Concurrent
- Managed Threading

Statement keywords (C# Reference)

Article • 03/30/2024

Statements are program instructions. Except as described in the topics referenced in the following table, statements are executed in sequence. The following table lists the C# statement keywords. For more information about statements that are not expressed with any keyword, see Statements.

Expand table

Category	C# keywords
Selection statements	`if`, `switch`
Iteration statements	`do`, `for`, `foreach`, `while`
Jump statements	`break`, `continue`, `goto`, `return`
Exception-handling statements	`throw`, `try-catch`, `try-finally`, `try-catch-finally`
checked and unchecked statements	`checked`, `unchecked`
fixed statement	`fixed`
lock statement	`lock`
yield statement	`yield`

See also

- Statements
- C# Keywords

Method parameters and modifiers

Article • 12/06/2024

By default, arguments in C# are passed to functions *by value*. That means a copy of the variable is passed to the method. For value (`struct`) types, a copy of the *value* is passed to the method. For reference (`class`) types, a copy of the *reference* is passed to the method. Parameter modifiers enable you to pass arguments *by reference*.

Because a struct is a value type, the method receives and operates on a copy of the argument when you pass a struct by value to a method. The method has no access to the original struct in the calling method and therefore can't change it in any way. The method can change only the copy.

A class instance is a reference type not a value type. When a reference type is passed by value to a method, the method receives a copy of the reference to the instance. Both variables refer to the same object. The parameter is a copy of the reference. The called method can't reassign the instance in the calling method. However, the called method can use the copy of the reference to access the instance members. If the called method changes an instance member, the calling method also sees those changes since it references the same instance.

Pass by value and pass by reference

All the examples in this section use the following two `record` types to illustrate the differences between `class` types and `struct` types:

C#

```
public record struct Point(int X, int Y);
// This doesn't use a primary constructor because the properties implemented
for `record` types are
// readonly in record class types. That would prevent the mutations
necessary for this example.
public record class Point3D
{
    public int X { get; set; }
    public int Y { get; set; }
    public int Z { get; set; }
}
```

The output of the following example illustrates the difference between passing a struct type by value and passing a class type by value. Both `Mutate` methods change property

values of its argument. When the parameter is a `struct` type, those changes are made to a copy of the argument's data. When the parameter is a `class` type, those changes are made to the instance referred to by the argument:

C#

```
public class PassTypesByValue
{
    public static void Mutate(Point pt)
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }
    public static void Mutate(Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;
        pt.Z = 42;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }

    public static void TestPassTypesByValue()
    {
        Console.WriteLine("===== Value Types =====");

        var ptStruct = new Point { X = 1, Y = 2 };
        Console.WriteLine($"After initialization:\t\t{ptStruct}");

        Mutate(ptStruct);

        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{ptStruct}");

        Console.WriteLine("===== Reference Types =====");

        var ptClass = new Point3D { X = 1, Y = 2, Z = 3 };

        Console.WriteLine($"After initialization:\t\t{ptClass}");

        Mutate(ptClass);
        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{ptClass}");

        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Mutate:           Point { X = 1, Y = 2 }
        //         Exit Mutate:            Point { X = 19, Y = 23 }
        // After called Mutate:            Point { X = 1, Y = 2 }
        // ===== Reference Types =====
```

```
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Mutate:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
        // After called Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
    }
}
```

The `ref` modifier is one way to pass arguments *by reference* to methods. The following code follows the preceding example, but passes parameters by reference. The modifications made to the `struct` type are visible in the calling method when the struct is passed by reference. There's no semantic change when a reference type is passed by reference:

C#

```
public class PassTypesByReference
{
    public static void Mutate(ref Point pt)
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }
    public static void Mutate(ref Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Mutate)}:\t\t{pt}");
        pt.X = 19;
        pt.Y = 23;
        pt.Z = 42;

        Console.WriteLine($"\tExit {nameof(Mutate)}:\t\t{pt}");
    }

    public static void TestPassTypesByReference()
    {
        Console.WriteLine("===== Value Types =====");

        var pStruct = new Point { X = 1, Y = 2 };
        Console.WriteLine($"After initialization:\t\t{pStruct}");

        Mutate(ref pStruct);

        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{pStruct}");

        Console.WriteLine("===== Reference Types =====");

        var pClass = new Point3D { X = 1, Y = 2, Z = 3 };
```

```
        Console.WriteLine($"After initialization:\t\t{pClass}");

        Mutate(ref pClass);
        Console.WriteLine($"After called {nameof(Mutate)}:\t\t{pClass}");

        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Mutate:           Point { X = 1, Y = 2 }
        //         Exit Mutate:            Point { X = 19, Y = 23 }
        // After called Mutate:            Point { X = 19, Y = 23 }
        // ===== Reference Types =====
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Mutate:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
        // After called Mutate:            Point3D { X = 19, Y = 23, Z = 42
}
    }
}
```

The preceding examples modified properties of a parameter. A method can also reassign a parameter to a new value. Reassignment behaves differently for struct and class types when passed by value or by reference. The following example shows how struct types and class types behave when parameters that are passed by value are reassigned:

C#

```
public class PassByValueReassignment
{
    public static void Reassign(Point pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point { X = 13, Y = 29 };

        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void Reassign(Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point3D { X = 13, Y = 29, Z = -42 };

        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void TestPassByValueReassignment()
    {
        Console.WriteLine("===== Value Types =====");

        var ptStruct = new Point { X = 1, Y = 2 };
```

```
        Console.WriteLine($"After initialization:\t\t{ptStruct}");

        Reassign(ptStruct);

        Console.WriteLine($"After called
{nameof(Reassign)}:\t\t{ptStruct}");

        Console.WriteLine("===== Reference Types =====");

        var ptClass = new Point3D { X = 1, Y = 2, Z = 3 };

        Console.WriteLine($"After initialization:\t\t{ptClass}");

        Reassign(ptClass);
        Console.WriteLine($"After called {nameof(Reassign)}:\t\t{ptClass}");

        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Reassign:         Point { X = 1, Y = 2 }
        //         Exit Reassign:          Point { X = 13, Y = 29 }
        // After called Reassign:          Point { X = 1, Y = 2 }
        // ===== Reference Types =====
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Reassign:         Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Reassign:          Point3D { X = 13, Y = 29, Z = -42
}
        // After called Reassign:          Point3D { X = 1, Y = 2, Z = 3 }
    }
}
```

The preceding sample shows that when you reassign a parameter to a new value, that change isn't visible from the calling method, regardless of whether the type is a value type or a reference type. The following example shows the behavior when you reassign a parameter that has been passed by reference:

C#

```
public class PassByReferenceReassignment
{
    public static void Reassign(ref Point pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point { X = 13, Y = 29 };

        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void Reassign(ref Point3D pt)
    {
        Console.WriteLine($"\tEnter {nameof(Reassign)}:\t\t{pt}");
        pt = new Point3D { X = 13, Y = 29, Z = -42 };
```

```
        Console.WriteLine($"\tExit {nameof(Reassign)}:\t\t{pt}");
    }

    public static void TestPassByReferenceReassignment()
    {
        Console.WriteLine("===== Value Types =====");

        var ptStruct = new Point { X = 1, Y = 2 };
        Console.WriteLine($"After initialization:\t\t{ptStruct}");

        Reassign(ref ptStruct);

        Console.WriteLine($"After called
{nameof(Reassign)}:\t\t{ptStruct}");

        Console.WriteLine("===== Reference Types =====");

        var ptClass = new Point3D { X = 1, Y = 2, Z = 3 };

        Console.WriteLine($"After initialization:\t\t{ptClass}");

        Reassign(ref ptClass);
        Console.WriteLine($"After called {nameof(Reassign)}:\t\t{ptClass}");

        // Output:
        // ===== Value Types =====
        // After initialization:           Point { X = 1, Y = 2 }
        //         Enter Reassign:         Point { X = 1, Y = 2 }
        //         Exit Reassign:          Point { X = 13, Y = 29 }
        // After called Reassign:          Point { X = 13, Y = 29 }
        // ===== Reference Types =====
        // After initialization:           Point3D { X = 1, Y = 2, Z = 3 }
        //         Enter Reassign:         Point3D { X = 1, Y = 2, Z = 3 }
        //         Exit Reassign:          Point3D { X = 13, Y = 29, Z = -42
}
        // After called Reassign:          Point3D { X = 13, Y = 29, Z = -42
}
    }
}
```

The preceding example shows how reassigning the value of a parameter that is passed by reference is visible in the calling context.

Safe context of references and values

Methods can store the values of parameters in fields. When parameters are passed by value, that's usually safe. Values are copied, and reference types are reachable when stored in a field. Passing parameters by reference safely requires the compiler to define when it's safe to assign a reference to a new variable. For every expression, the compiler

defines a *safe context* that bounds access to an expression or variable. The compiler uses two scopes: *safe-context* and *ref-safe-context*.

- The *safe-context* defines the scope where any expression can be safely accessed.
- The *ref-safe-context* defines the scope where a *reference* to any expression can be safely accessed or modified.

Informally, you can think of these scopes as the mechanism to ensure your code never accesses or modifies a reference that's no longer valid. A reference is valid as long as it refers to a valid object or struct. The *safe-context* defines when a variable can be assigned or reassigned. The *ref-safe-context* defines when a variable can be *ref assigned* or *ref reassigned*. Assignment assigns a variable to a new value; *ref assignment* assigns the variable to *refer to* a different storage location.

Reference parameters

You apply one of the following modifiers to a parameter declaration to pass arguments by reference instead of by value:

- ref: The argument must be initialized before calling the method. The method can assign a new value to the parameter, but isn't required to do so.
- out: The calling method isn't required to initialize the argument before calling the method. The method must assign a value to the parameter.
- ref readonly: The argument must be initialized before calling the method. The method can't assign a new value to the parameter.
- in: The argument must be initialized before calling the method. The method can't assign a new value to the parameter. The compiler might create a temporary variable to hold a copy of the argument to `in` parameters.

A parameter that is passed by reference is a *reference variable*. It doesn't have it's own value. Instead, it refers to a different variable called its *referrent*. Reference variables can be ref reassigned, which changes its referrent.

Members of a class can't have signatures that differ only by `ref`, `ref readonly`, `in`, or `out`. A compiler error occurs if the only difference between two members of a type is that one of them has a `ref` parameter and the other has an `out`, `ref readonly`, or `in` parameter. However, methods can be overloaded when one method has a `ref`, `ref readonly`, `in`, or `out` parameter and the other has a parameter that is passed by value, as shown in the following example. In other situations that require signature matching, such as hiding or overriding, `in`, `ref`, `ref readonly`, and `out` are part of the signature and don't match each other.

When a parameter has one of the preceding modifiers, the corresponding argument can have a compatible modifier:

- An argument for a `ref` parameter must include the `ref` modifier.
- An argument for an `out` parameter must include the `out` modifier.
- An argument for an `in` parameter can optionally include the `in` modifier. If the `ref` modifier is used on the argument instead, the compiler issues a warning.
- An argument for a `ref readonly` parameter should include either the `in` or `ref` modifiers, but not both. If neither modifier is included, the compiler issues a warning.

When you use these modifiers, they describe how the argument is used:

- `ref` means the method can read or write the value of the argument.
- `out` means the method sets the value of the argument.
- `ref readonly` means the method reads, but can't write the value of the argument. The argument *should* be passed by reference.
- `in` means the method reads, but can't write the value of the argument. The argument will be passed by reference or through a temporary variable.

You can't use the previous parameter modifiers in the following kinds of methods:

- Async methods, which you define by using the async modifier.
- Iterator methods, which include a yield return or `yield break` statement.

Extension methods also have restrictions on the use of these argument keywords:

- The `out` keyword can't be used on the first argument of an extension method.
- The `ref` keyword can't be used on the first argument of an extension method when the argument isn't a `struct`, or a generic type not constrained to be a struct.
- The `ref readonly` and `in` keywords can't be used unless the first argument is a `struct`.
- The `ref readonly` and `in` keywords can't be used on any generic type, even when constrained to be a struct.

Properties aren't variables. They're methods. Properties can't be arguments for `ref` parameters.

`ref` parameter modifier

To use a `ref` parameter, both the method definition and the calling method must explicitly use the `ref` keyword, as shown in the following example. (Except that the

calling method can omit `ref` when making a COM call.)

```C#
void Method(ref int refArgument)
{
    refArgument = refArgument + 44;
}

int number = 1;
Method(ref number);
Console.WriteLine(number);
// Output: 45
```

An argument that is passed to a `ref` parameter must be initialized before it's passed.

`out` parameter modifier

To use an `out` parameter, both the method definition and the calling method must explicitly use the `out` keyword. For example:

```C#
int initializeInMethod;
OutArgExample(out initializeInMethod);
Console.WriteLine(initializeInMethod);     // value is now 44

void OutArgExample(out int number)
{
    number = 44;
}
```

Variables passed as `out` arguments don't have to be initialized before being passed in a method call. However, the called method is required to assign a value before the method returns.

Deconstruct methods declare their parameters with the `out` modifier to return multiple values. Other methods can return value tuples for multiple return values.

You can declare a variable in a separate statement before you pass it as an `out` argument. You can also declare the `out` variable in the argument list of the method call, rather than in a separate variable declaration. `out` variable declarations produce more compact, readable code, and also prevent you from inadvertently assigning a value to the variable before the method call. The following example defines the `number` variable in the call to the Int32.TryParse method.

C#

```
string numberAsString = "1640";

if (Int32.TryParse(numberAsString, out int number))
    Console.WriteLine($"Converted '{numberAsString}' to {number}");
else
    Console.WriteLine($"Unable to convert '{numberAsString}'");
// The example displays the following output:
//       Converted '1640' to 1640
```

You can also declare an implicitly typed local variable.

`ref readonly` modifier

The `ref readonly` modifier must be present in the method declaration. A modifier at the call site is optional. Either the `in` or `ref` modifier can be used. The `ref readonly` modifier isn't valid at the call site. Which modifier you use at the call site can help describe characteristics of the argument. You can only use `ref` if the argument is a variable, and is writable. You can only use `in` when the argument is a variable. It might be writable, or readonly. You can't add either modifier if the argument isn't a variable, but is an expression. The following examples show these conditions. The following method uses the `ref readonly` modifier to indicate that a large struct should be passed by reference for performance reasons:

C#

```
public static void ForceByRef(ref readonly OptionStruct thing)
{
    // elided
}
```

You can call the method using the `ref` or `in` modifier. If you omit the modifier, the compiler issues a warning. When the argument is an expression, not a variable, you can't add the `in` or `ref` modifiers, so you should suppress the warning:

C#

```
ForceByRef(in options);
ForceByRef(ref options);
ForceByRef(options); // Warning! variable should be passed with `ref` or
`in`
ForceByRef(new OptionStruct()); // Warning, but an expression, so no
variable to reference
```

If the variable is a `readonly` variable, you must use the `in` modifier. The compiler issues an error if you use the `ref` modifier instead.

The `ref readonly` modifier indicates that the method expects the argument to be a variable rather than an expression that isn't a variable. Examples of expressions that aren't variables are constants, method return values, and properties. If the argument isn't a variable, the compiler issues a warning.

`in` parameter modifier

The `in` modifier is required in the method declaration but unnecessary at the call site.

C#

```
int readonlyArgument = 44;
InArgExample(readonlyArgument);
Console.WriteLine(readonlyArgument);     // value is still 44

void InArgExample(in int number)
{
    // Uncomment the following line to see error CS8331
    //number = 19;
}
```

The `in` modifier allows the compiler to create a temporary variable for the argument and pass a readonly reference to that argument. The compiler always creates a temporary variable when the argument must be converted, when there's an implicit conversion from the argument type, or when the argument is a value that isn't a variable. For example, when the argument is a literal value, or the value returned from a property accessor. When your API requires that the argument be passed by reference, choose the `ref readonly` modifier instead of the `in` modifier.

Methods that are defined using `in` parameters potentially gain performance optimization. Some `struct` type arguments might be large in size, and when methods are called in tight loops or critical code paths, the cost of copying those structures is substantial. Methods declare `in` parameters to specify that arguments can be passed by reference safely because the called method doesn't modify the state of that argument. Passing those arguments by reference avoids the (potentially) expensive copy. You explicitly add the `in` modifier at the call site to ensure the argument is passed by reference, not by value. Explicitly using `in` has the following two effects:

- Specifying `in` at the call site forces the compiler to select a method defined with a matching `in` parameter. Otherwise, when two methods differ only in the presence

of `in`, the by value overload is a better match.

- By specifying `in`, you declare your intent to pass an argument by reference. The argument used with `in` must represent a location that can be directly referred to. The same general rules for `out` and `ref` arguments apply: You can't use constants, ordinary properties, or other expressions that produce values. Otherwise, omitting `in` at the call site informs the compiler that it's fine to create a temporary variable to pass by read-only reference to the method. The compiler creates a temporary variable to overcome several restrictions with `in` arguments:
  - A temporary variable allows compile-time constants as `in` parameters.
  - A temporary variable allows properties, or other expressions for `in` parameters.
  - A temporary variable allows arguments where there's an implicit conversion from the argument type to the parameter type.

In all the preceding instances, the compiler creates a temporary variable that stores the value of the constant, property, or other expression.

The following code illustrates these rules:

C#

```
static void Method(in int argument)
{
    // implementation removed
}

Method(5); // OK, temporary variable created.
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // OK, temporary int created with the value 0
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // passed by readonly reference
Method(in i); // passed by readonly reference, explicitly using `in`
```

Now, suppose another method using by-value arguments was available. The results change as shown in the following code:

C#

```
static void Method(int argument)
{
    // implementation removed
}

static void Method(in int argument)
{
    // implementation removed
```

```
}

Method(5); // Calls overload passed by value
Method(5L); // CS1503: no implicit conversion from long to int
short s = 0;
Method(s); // Calls overload passed by value.
Method(in s); // CS1503: cannot convert from in short to in int
int i = 42;
Method(i); // Calls overload passed by value
Method(in i); // passed by readonly reference, explicitly using `in`
```

The only method call where the argument is passed by reference is the final one.

> **Note**
>
> The preceding code uses `int` as the argument type for simplicity. Because `int` is no larger than a reference in most modern machines, there is no benefit to passing a single `int` as a readonly reference.

`params` modifier

No other parameters are permitted after the `params` keyword in a method declaration, and only one `params` keyword is permitted in a method declaration.

The declared type of the `params` parameter must be a collection type. Recognized collection types are:

- A single dimensional *array type* `T[]`, in which case the *element type* is `T`.
- A *span type*:
  - `System.Span<T>`
  - `System.ReadOnlySpan<T>`
    Here, the *element type* is `T`.
- A *type* with an accessible *create method* with a corresponding *element type*. The *create method* is identified using the same attribute used for collection expressions.
- A *struct* or *class type* that implements System.Collections.Generic.IEnumerable<T> where:
  - The *type* has a constructor that can be invoked with no arguments, and the constructor is at least as accessible as the declaring member.
  - The *type* has an instance (not an extension) method `Add` where:
    - The method can be invoked with a single value argument.

    - If the method is generic, the type arguments can be inferred from the argument.
    - The method is at least as accessible as the declaring member. Here, the *element type* is the *iteration type* of the *type*.
- An *interface type*:
    - System.Collections.Generic.IEnumerable<T>
    - System.Collections.Generic.IReadOnlyCollection<T>
    - System.Collections.Generic.IReadOnlyList<T>
    - System.Collections.Generic.ICollection<T>
    - System.Collections.Generic.IList<T> Here, the *element type* is `T`.

Before C# 13, the parameter must be a single dimensional array.

When you call a method with a `params` parameter, you can pass in:

- A comma-separated list of arguments of the type of the array elements.
- A collection of arguments of the specified type.
- No arguments. If you send no arguments, the length of the `params` list is zero.

The following example demonstrates various ways in which arguments can be sent to a `params` parameter.

C#

```
public static void ParamsModifierExample(params int[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        System.Console.Write(list[i] + " ");
    }
    System.Console.WriteLine();
}

public static void ParamsModifierObjectExample(params object[] list)
{
    for (int i = 0; i < list.Length; i++)
    {
        System.Console.Write(list[i] + " ");
    }
    System.Console.WriteLine();
}

public static void TryParamsCalls()
{
    // You can send a comma-separated list of arguments of the
    // specified type.
    ParamsModifierExample(1, 2, 3, 4);
    ParamsModifierObjectExample(1, 'a', "test");
```

```
    // A params parameter accepts zero or more arguments.
    // The following calling statement displays only a blank line.
    ParamsModifierObjectExample();

    // An array argument can be passed, as long as the array
    // type matches the parameter type of the method being called.
    int[] myIntArray = { 5, 6, 7, 8, 9 };
    ParamsModifierExample(myIntArray);

    object[] myObjArray = { 2, 'b', "test", "again" };
    ParamsModifierObjectExample(myObjArray);

    // The following call causes a compiler error because the object
    // array cannot be converted into an integer array.
    //ParamsModifierExample(myObjArray);

    // The following call does not cause an error, but the entire
    // integer array becomes the first element of the params array.
    ParamsModifierObjectExample(myIntArray);
}
/*
Output:
    1 2 3 4
    1 a test

    5 6 7 8 9
    2 b test again
    System.Int32[]
*/
```

Overload resolution can cause ambiguity when the argument for a `params` parameter is a collection type. The collection type of the argument must be convertible to the collection type of the parameter. When different overloads provide better conversions for that parameter, that method might be better. However, if the argument to the `params` parameter is either discrete elements or missing, all overloads with different `params` parameter types are equal for that parameter.

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

The `namespace` keyword

Article • 11/22/2024

The `namespace` keyword is used to declare a scope that contains a set of related objects. You can use a namespace to organize code elements and to create globally unique types.

C#

```
namespace SampleNamespace
{
    class SampleClass { }

    interface ISampleInterface { }

    struct SampleStruct { }

    enum SampleEnum { a, b }

    delegate void SampleDelegate(int i);

    namespace Nested
    {
        class SampleClass2 { }
    }
}
```

File scoped namespace declarations enable you to declare that all types in a file are in a single namespace. The following example is similar to the previous example, but uses a file scoped namespace declaration:

C#

```
using System;

namespace SampleFileScopedNamespace;

class SampleClass { }

interface ISampleInterface { }

struct SampleStruct { }

enum SampleEnum { a, b }

delegate void SampleDelegate(int i);
```

Using Statements in File Scoped Namespaces

When you use *file-scoped namespaces*, the placement of `using` statements affects their scope within the file. File-scoped namespaces lower to the equivalent traditional namespace declaration that ends with a closing bracket at the end of the file. This behavior determines where `using` directives are applied as follows:

- If the `using` statements are placed before the file-scoped namespace declaration, they're treated as being outside of the namespace and are interpreted as fully qualified namespaces.
- If the `using` statements are placed after the file-scoped namespace declaration, they're scoped within the namespace itself.

For example:

C#

```
// This using is outside the namespace scope, so it applies globally
using System;

namespace SampleNamespace; // File-scoped namespace declaration

// This using is inside the namespace scope
using System.Text;

public class SampleClass
{
    // Class members...
}
```

In the preceding example, `System` is globally accessible, while `System.Text` applies only within `SampleNamespace`.

The preceding example doesn't include a nested namespace. File scoped namespaces can't include more namespace declarations. You can't declare a nested namespace or a second file-scoped namespace:

C#

```
namespace SampleNamespace;

class AnotherSampleClass
{
    public void AnotherSampleMethod()
    {
        System.Console.WriteLine(
            "SampleMethod inside SampleNamespace");
```

```
    }
}

namespace AnotherNamespace; // Not allowed!

namespace ANestedNamespace // Not allowed!
{
   // declarations...
}
```

Within a namespace, you can declare zero or more of the following types:

- class
- interface
- struct
- enum
- delegate
- nested namespaces can be declared except in file scoped namespace declarations

The compiler adds a default namespace. This unnamed namespace, sometimes referred to as the global namespace, is present in every file. It contains declarations not included in a declared namespace. Any identifier in the global namespace is available for use in a named namespace.

Namespaces implicitly have public access. For a discussion of the access modifiers you can assign to elements in a namespace, see Access Modifiers.

It's possible to define a namespace in two or more declarations. For example, the following example defines two classes as part of the `MyCompany` namespace:

C#

```
namespace MyCompany.Proj1
{
    class MyClass
    {
    }
}

namespace MyCompany.Proj1
{
    class MyClass1
    {
    }
}
```

The following example shows how to call a static method in a nested namespace.

C#

```
namespace SomeNameSpace
{
    public class MyClass
    {
        static void Main()
        {
            Nested.NestedNameSpaceClass.SayHello();
        }
    }

    // a nested namespace
    namespace Nested
    {
        public class NestedNameSpaceClass
        {
            public static void SayHello()
            {
                Console.WriteLine("Hello");
            }
        }
    }
}
// Output: Hello
```

C# language specification

For more information, see the Namespaces section of the C# language specification. For more information on file scoped namespace declarations, see the feature specification.

See also

- Namespace declaration preferences (IDE0160 and IDE0161)
- C# keywords
- using
- using static
- Namespace alias qualifier ::
- Namespaces

The `using` directive

Article • 01/27/2025

The `using` directive allows you to use types defined in a namespace without specifying the fully qualified namespace of that type. In its basic form, the `using` directive imports all the types from a single namespace, as shown in the following example:

```
C#

using System.Text;
```

You can apply two modifiers to a `using` directive:

- The `global` modifier has the same effect as adding the same `using` directive to every source file in your project.
- The `static` modifier imports the `static` members and nested types from a single type rather than importing all the types in a namespace.

You can combine both modifiers to import the static members from a type to all source files in your project.

You can also create an alias for a namespace or a type with a *using alias directive*.

```
C#

using Project = PC.MyCompany.Project;
```

You can use the `global` modifier on a *using alias directive*.

> **Note**
>
> The `using` keyword is also used to create *`using` statements*, which help ensure that IDisposable objects such as files and fonts are handled correctly. For more information about the *`using` statement*, see using statement.

The scope of a `using` directive without the `global` modifier is the file in which it appears.

The `global using` directive must appear before all namespace and type declarations. All global using directives must appear in a source file before any nonglobal `using` directives.

Other `using` directives can appear:

- At the beginning of a source code file, before any namespace or type declarations.
- In any blocked-scoped namespace, but before any namespaces or types declared in that namespace.

Otherwise, a compiler error is generated.

Create a `using` directive to use the types in a namespace without having to specify the namespace. A `using` directive doesn't give you access to any namespaces that are nested in the namespace you specify. Namespaces come in two categories: user-defined and system-defined. User-defined namespaces are namespaces defined in your code. For a list of the system-defined namespaces, see .NET API Browser.

The `global` modifier

Adding the `global` modifier to a `using` directive means that using is applied to all files in the compilation (typically a project):

```
C#

global using <fully-qualified-namespace>;
```

Where *fully-qualified-namespace* is the fully qualified name of the namespace whose types can be referenced without specifying the namespace.

A *global using* directive can appear at the beginning of any source code file. All `global using` directives in a single file must appear before:

- All `using` directives without the `global` modifier.
- All namespace and type declarations in the file.

You can add `global using` directives to any source file. Typically, you want to keep them in a single location. The order of `global using` directives doesn't matter, either in a single file, or between files.

The `global` modifier can be combined with the `static` modifier. The `global` modifier can be applied to a *using alias directive*. In both cases, the directive's scope is all files in the current compilation. The following example enables using all the methods declared in the System.Math in all files in your project:

```
C#
```

```
global using static System.Math;
```

You can also globally include a namespace by adding a `<Using>` item to your project file, for example, `<Using Include="My.Awesome.Namespace" />`. For more information, see <Using> item.

Analyzers issue diagnostics if you duplicate `global` using directives in different locations. These same analyzers also inform you if you add a `using` directive for a namespace or type that a `global` using directive already references. You might find it easier to manage your `global` usings by keeping them together in one file in the project.

> **ⓘ 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

The `static` modifier

The `using static` directive names a type whose static members and nested types you can access without specifying a type name. Its syntax is:

```
C#

using static <fully-qualified-type-name>;
```

The `<fully-qualified-type-name>` is the name of the type whose static members and nested types can be referenced without specifying a type name. If you don't provide a fully qualified type name (the full namespace name along with the type name), C# generates compiler error CS0246: "The type or namespace name 'type/namespace' couldn't be found (are you missing a using directive or an assembly reference?)".

The `using static` directive applies to any type that has static members (or nested types), even if it also has instance members. However, instance members can only be invoked through the type instance.

You can access static members of a type without having to qualify the access with the type name:

C#

```
using static System.Console;
using static System.Math;
class Program
{
    static void Main()
    {
        WriteLine(Sqrt(3*3 + 4*4));
    }
}
```

Ordinarily, when you call a static member, you provide the type name along with the member name. Repeatedly entering the same type name to invoke members of the type can result in verbose, obscure code. For example, the following definition of a `Circle` class references many members of the Math class.

C#

```
using System;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }
```

```
    public double Circumference
    {
       get { return 2 * Radius * Math.PI; }
    }

    public double Area
    {
       get { return Math.PI * Math.Pow(Radius, 2); }
    }
}
```

By eliminating the need to explicitly reference the Math class each time a member is referenced, the `using static` directive produces cleaner code:

C#

```
using System;
using static System.Math;

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
   {
      get { return 2 * Radius; }
   }

   public double Circumference
   {
      get { return 2 * Radius * PI; }
   }

   public double Area
   {
      get { return PI * Pow(Radius, 2); }
   }
}
```

`using static` imports only accessible static members and nested types declared in the specified type. Inherited members aren't imported. You can import from any named type with a `using static` directive, including Visual Basic modules. If F# top-level functions appear in metadata as static members of a named type whose name is a valid C# identifier, then the F# functions can be imported.

`using static` makes extension methods declared in the specified type available for extension method lookup. However, the names of the extension methods aren't imported into scope for unqualified reference in code.

Methods with the same name imported from different types by different `using static` directives in the same compilation unit or namespace form a method group. Overload resolution within these method groups follows normal C# rules.

The following example uses the `using static` directive to make the static members of the Console, Math, and String classes available without having to specify their type name.

C#

```
using System;
using static System.Console;
using static System.Math;
using static System.String;

class Program
{
   static void Main()
   {
      Write("Enter a circle's radius: ");
      var input = ReadLine();
      if (!IsNullOrEmpty(input) && double.TryParse(input, out var radius)) {
         var c = new Circle(radius);

         string s = "\nInformation about the circle:\n";
         s = s + Format("   Radius: {0:N2}\n", c.Radius);
         s = s + Format("   Diameter: {0:N2}\n", c.Diameter);
         s = s + Format("   Circumference: {0:N2}\n", c.Circumference);
         s = s + Format("   Area: {0:N2}\n", c.Area);
         WriteLine(s);
      }
      else {
         WriteLine("Invalid input...");
      }
   }
}

public class Circle
{
   public Circle(double radius)
   {
      Radius = radius;
   }

   public double Radius { get; set; }

   public double Diameter
```

```
    {
        get { return 2 * Radius; }
    }

    public double Circumference
    {
        get { return 2 * Radius * PI; }
    }

    public double Area
    {
        get { return PI * Pow(Radius, 2); }
    }
}
// The example displays the following output:
//       Enter a circle's radius: 12.45
//
//       Information about the circle:
//          Radius: 12.45
//          Diameter: 24.90
//          Circumference: 78.23
//          Area: 486.95
```

In the example, the `using static` directive could also be applied to the Double type. Adding that directive would make it possible to call the TryParse(String, Double) method without specifying a type name. However, using `TryParse` without a type name creates less readable code, since it becomes necessary to check the `using static` directives to determine which numeric type's `TryParse` method is called.

`using static` also applies to `enum` types. By adding `using static` with the enum, the type is no longer required to use the enum members.

C#

```
using static Color;

enum Color
{
    Red,
    Green,
    Blue
}

class Program
{
    public static void Main()
    {
        Color color = Green;
    }
}
```

The `using` alias

Create a `using` alias directive to make it easier to qualify an identifier to a namespace or type. In any `using` directive, the fully qualified namespace or type must be used regardless of the `using` directives that come before it. No `using` alias can be used in the declaration of a `using` directive. For example, the following example generates a compiler error:

C#

```
using s = System.Text;
using s.RegularExpressions; // Generates a compiler error.
```

The following example shows how to define and use a `using` alias for a namespace:

C#

```
namespace PC
{
    // Define an alias for the nested namespace.
    using Project = PC.MyCompany.Project;
    class A
    {
        void M()
        {
            // Use the alias
            var mc = new Project.MyClass();
        }
    }
    namespace MyCompany
    {
        namespace Project
        {
            public class MyClass { }
        }
    }
}
```

A using alias directive can't have an open generic type on the right-hand side. For example, you can't create a using alias for a `List<T>`, but you can create one for a `List<int>`.

The following example shows how to define a `using` directive and a `using` alias for a class:

C#

```
using System;

// Using alias directive for a class.
using AliasToMyClass = NameSpace1.MyClass;

// Using alias directive for a generic class.
using UsingAlias = NameSpace2.MyClass<int>;

namespace NameSpace1
{
    public class MyClass
    {
        public override string ToString()
        {
            return "You are in NameSpace1.MyClass.";
        }
    }
}

namespace NameSpace2
{
    class MyClass<T>
    {
        public override string ToString()
        {
            return "You are in NameSpace2.MyClass.";
        }
    }
}

namespace NameSpace3
{
    class MainClass
    {
        static void Main()
        {
            var instance1 = new AliasToMyClass();
            Console.WriteLine(instance1);

            var instance2 = new UsingAlias();
            Console.WriteLine(instance2);
        }
    }
}
// Output:
//    You are in NameSpace1.MyClass.
//    You are in NameSpace2.MyClass.
```

Beginning with C# 12, you can create aliases for types that were previously restricted, including tuple types, pointer types, and other unsafe types. For more information on the updated rules, see the feature spec.

Qualified alias member

The namespace alias qualifier, `::` provides explicit access to the global namespace or other using aliases potentially hidden by other entities.

The `global::` ensures that the namespace lookup for the namespace following the `::` token is relative to the global namespace. Otherwise, the token must resolve to a using alias, and the token following the `::` must resolve to a type in that aliased namespace. The following example shows both forms:

C#

```
using S = System.Net.Sockets;

class A
{
    public static int x;
}

class C
{
    public void F(int A, object S)
    {
        // Use global::A.x instead of A.x
        global::A.x += A;

        // Using ::, S must resolve to a namespace alias:
        S::Socket s = S as S::Socket;

        // In this form, if S were a class, it would be a compile-time
error:
        S.Socket s1 = S as S.Socket;
    }
}
```

C# language specification

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

For more information on the *global using* modifier, see the global usings feature specification.

See also

- C# keywords

- Namespaces
- Style rule IDE0005 - Remove unnecessary 'using' directives
- Style rule IDE0065 - 'using' directive placement
- using statement

extern alias (C# Reference)

Article • 03/30/2024

You might have to reference two versions of assemblies that have the same fully-qualified type names. For example, you might have to use two or more versions of an assembly in the same application. By using an external assembly alias, the namespaces from each assembly can be wrapped inside root-level namespaces named by the alias, which enables them to be used in the same file.

> **Note**
>
> The **extern** keyword is also used as a method modifier, declaring a method written in unmanaged code.

To reference two assemblies with the same fully-qualified type names, an alias must be specified at a command prompt, as follows:

```
/r:GridV1=grid.dll
```

```
/r:GridV2=grid20.dll
```

This creates the external aliases `GridV1` and `GridV2`. To use these aliases from within a program, reference them by using the `extern` keyword. For example:

```
extern alias GridV1;
```

```
extern alias GridV2;
```

Each extern alias declaration introduces an additional root-level namespace that parallels (but does not lie within) the global namespace. Thus types from each assembly can be referred to without ambiguity by using their fully qualified name, rooted in the appropriate namespace-alias.

In the previous example, `GridV1::Grid` would be the grid control from `grid.dll`, and `GridV2::Grid` would be the grid control from `grid20.dll`.

Using Visual Studio

If you are using Visual Studio, aliases can be provided in similar way.

Add reference of *grid.dll* and *grid20.dll* to your project in Visual Studio. Open a property tab and change the Aliases from global to GridV1 and GridV2 respectively.

Use these aliases the same way above

```
C#

 extern alias GridV1;

 extern alias GridV2;
```

Now you can create alias for a namespace or a type by *using alias directive*. For more information, see using directive.

```
C#

using Class1V1 = GridV1::Namespace.Class1;

using Class1V2 = GridV2::Namespace.Class1;
```

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

- C# Keywords
- :: Operator
- **References** (C# Compiler Options)

new constraint (C# Reference)

Article • 03/30/2024

The `new` constraint specifies that a type argument in a generic class or method declaration must have a public parameterless constructor. To use the `new` constraint, the type cannot be abstract.

Apply the `new` constraint to a type parameter when a generic class creates new instances of the type, as shown in the following example:

C#

```
class ItemFactory<T> where T : new()
{
    public T GetNewItem()
    {
        return new T();
    }
}
```

When you use the `new()` constraint with other constraints, it must be specified last:

C#

```
public class ItemFactory2<T>
    where T : IComparable, new()
{  }
```

For more information, see Constraints on Type Parameters.

You can also use the `new` keyword to create an instance of a type or as a member declaration modifier.

C# language specification

For more information, see the Type parameter constraints section of the C# language specification.

See also

- C# Keywords
- Generics

where (generic type constraint) (C# Reference)

Article • 07/30/2024

The `where` clause in a generic definition specifies constraints on the types that are used as arguments for type parameters in a generic type, method, delegate, or local function. Constraints can specify interfaces, base classes, or require a generic type to be a reference, value, or unmanaged type. They declare capabilities that the type argument must have, and must be placed after any declared base class or implemented interfaces.

For example, you can declare a generic class, `AGenericClass`, such that the type parameter `T` implements the IComparable<T> interface:

C#

```
public class AGenericClass<T> where T : IComparable<T> { }
```

> **Note**
>
> For more information on the where clause in a query expression, see where clause.

The `where` clause can also include a base class constraint. The base class constraint states that a type to be used as a type argument for that generic type has the specified class as a base class, or is that base class. If the base class constraint is used, it must appear before any other constraints on that type parameter. Some types are disallowed as a base class constraint: Object, Array, and ValueType. The following example shows the types that can now be specified as a base class:

C#

```
public class UsingEnum<T> where T : System.Enum { }

public class UsingDelegate<T> where T : System.Delegate { }

public class Multicaster<T> where T : System.MulticastDelegate { }
```

In a nullable context, the nullability of the base class type is enforced. If the base class is non-nullable (for example `Base`), the type argument must be non-nullable. If the base class is nullable (for example `Base?`), the type argument can be either a nullable or non-

nullable reference type. The compiler issues a warning if the type argument is a nullable reference type when the base class is non-nullable.

The `where` clause can specify that the type is a `class` or a `struct`. The `struct` constraint removes the need to specify a base class constraint of `System.ValueType`. The `System.ValueType` type can't be used as a base class constraint. The following example shows both the `class` and `struct` constraints:

C#

```
class MyClass<T, U>
    where T : class
    where U : struct
{ }
```

In a nullable context, the `class` constraint requires a type to be a non-nullable reference type. To allow nullable reference types, use the `class?` constraint, which allows both nullable and non-nullable reference types.

The `where` clause can include the `notnull` constraint. The `notnull` constraint limits the type parameter to non-nullable types. The type can be a value type or a non-nullable reference type. The `notnull` constraint is available for code compiled in a nullable enable context. Unlike other constraints, if a type argument violates the `notnull` constraint, the compiler generates a warning instead of an error. Warnings are only generated in a `nullable enable` context.

The addition of nullable reference types introduces a potential ambiguity in the meaning of `T?` in generic methods. If `T` is a `struct`, `T?` is the same as System.Nullable<T>. However, if `T` is a reference type, `T?` means that `null` is a valid value. The ambiguity arises because overriding methods can't include constraints. The new `default` constraint resolves this ambiguity. You add it when a base class or interface declares two overloads of a method, one that specifies the `struct` constraint, and one that doesn't have either the `struct` or `class` constraint applied:

C#

```
public abstract class B
{
    public void M<T>(T? item) where T : struct { }
    public abstract void M<T>(T? item);

}
```

You use the `default` constraint to specify that your derived class overrides the method without the constraint in your derived class, or explicit interface implementation. It's only valid on methods that override base methods, or explicit interface implementations:

```
C#
```

```
public class D : B
{
    // Without the "default" constraint, the compiler tries to override the
first method in B
    public override void M<T>(T? item) where T : default { }
}
```

> **Important**
>
> Generic declarations that include the `notnull` constraint can be used in a nullable oblivious context, but compiler does not enforce the constraint.

```
C#
```

```
#nullable enable
    class NotNullContainer<T>
        where T : notnull
    {
    }
#nullable restore
```

The `where` clause can also include an `unmanaged` constraint. The `unmanaged` constraint limits the type parameter to types known as unmanaged types. The `unmanaged` constraint makes it easier to write low-level interop code in C#. This constraint enables reusable routines across all unmanaged types. The `unmanaged` constraint can't be combined with the `class` or `struct` constraint. The `unmanaged` constraint enforces that the type must be a `struct`:

```
C#
```

```
class UnManagedWrapper<T>
    where T : unmanaged
{ }
```

The `where` clause can also include a constructor constraint, `new()`. That constraint makes it possible to create an instance of a type parameter using the `new` operator. The new()

Constraint lets the compiler know that any type argument supplied must have an accessible parameterless constructor. For example:

```
C#

public class MyGenericClass<T> where T : IComparable<T>, new()
{
    // The following line is not possible without new() constraint:
    T item = new T();
}
```

The `new()` constraint appears last in the `where` clause, unless it's followed by the `allows ref struct` anti-constraint. The `new()` constraint can't be combined with the `struct` or `unmanaged` constraints. All types satisfying those constraints must have an accessible parameterless constructor, making the `new()` constraint redundant.

This anti-constraint declares that the type argument for `T` can be a `ref struct` type. For example:

```
C#

public class GenericRefStruct<T> where T : allows ref struct
{
    // Scoped is allowed because T might be a ref struct
    public void M(scoped T parm)
    {

    }
}
```

The generic type or method must obey ref safety rules for any instance of `T` because it might be a `ref struct`. The `allows ref struct` clause can't be combined with the `class` or `class?` constraint. The `allows ref struct` anti-constraint must follow all constraints for that type argument.

With multiple type parameters, use one `where` clause for each type parameter, for example:

```
C#

public interface IMyInterface { }

namespace CodeExample
{
    class Dictionary<TKey, TVal>
        where TKey : IComparable<TKey>
```

```
        where TVal : IMyInterface
    {
        public void Add(TKey key, TVal val) { }
    }
}
```

You can also attach constraints to type parameters of generic methods, as shown in the following example:

C#

```
public void MyMethod<T>(T t) where T : IMyInterface { }
```

Notice that the syntax to describe type parameter constraints on delegates is the same as that of methods:

C#

```
delegate T MyDelegate<T>() where T : new();
```

For information on generic delegates, see Generic Delegates.

For details on the syntax and use of constraints, see Constraints on Type Parameters.

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

- Introduction to Generics
- new Constraint
- Constraints on Type Parameters

The base keyword

Article • 04/19/2025

The `base` keyword is used to access members of the base class from within a derived class. Use it if you want to:

- Call a method on the base class overridden by another method.
- Specify which base-class constructor should be called when creating instances of the derived class.

The base class access is permitted only in a constructor, in an instance method, and in an instance property accessor. Using the `base` keyword from within a static method produces an error.

The base class that is accessed is the base class specified in the class declaration. For example, if you specify `class ClassB : ClassA`, the members of ClassA are accessed from ClassB, regardless of the base class of ClassA.

In this example, both the base class `Person` and the derived class `Employee` have a method named `GetInfo`. By using the `base` keyword, it's possible to call the `GetInfo` method of the base class from within the derived class.

C#

```
public class Person
{
    protected string ssn = "444-55-6666";
    protected string name = "John L. Malgraine";

    public virtual void GetInfo()
    {
        Console.WriteLine($"Name: {name}");
        Console.WriteLine($"SSN: {ssn}");
    }
}
class Employee : Person
{
    public readonly string id = "ABC567EFG";
    public override void GetInfo()
    {
        // Calling the base class GetInfo method:
        base.GetInfo();
        Console.WriteLine($"Employee ID: {id}");
    }
}

class TestClass
{
    static void Main()
```

```
    {
        Employee E = new Employee();
        E.GetInfo();
    }
}
/*
Output
Name: John L. Malgraine
SSN: 444-55-6666
Employee ID: ABC567EFG
*/
```

This example shows how to specify the base-class constructor called when creating instances of a derived class.

C#

```
public class BaseClass
{
    private int num;

    public BaseClass() =>
        Console.WriteLine("in BaseClass()");

    public BaseClass(int i)
    {
        num = i;
        Console.WriteLine("in BaseClass(int i)");
    }

    public int GetNum() => num;
}

public class DerivedClass : BaseClass
{
    // This constructor will call BaseClass.BaseClass()
    public DerivedClass() : base() { }

    // This constructor will call BaseClass.BaseClass(int i)
    public DerivedClass(int i) : base(i) { }

    static void Main()
    {
        DerivedClass md = new DerivedClass();
        DerivedClass md1 = new DerivedClass(1);
    }
}
/*
Output:
in BaseClass()
in BaseClass(int i)
*/
```

For more examples, see new, virtual, and override.

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

- C# Keywords
- The this keyword

The this keyword

Article • 04/19/2025

The `this` keyword refers to the current instance of the class and is also used as a modifier of the first parameter of an extension method.

> **Note**
>
> This article discusses the use of `this` with class instances. For more information about its use in extension methods, see the **extension** keyword.

The following are common uses of `this`:

- To qualify members hidden by similar names, for example:

```
C#

public class Employee
{
    private string alias;
    private string name;

    public Employee(string name, string alias)
    {
        // Use this to qualify the members of the class
        // instead of the constructor parameters.
        this.name = name;
        this.alias = alias;
    }
}
```

- To pass an object as a parameter to other methods, for example:

```
C#

CalcTax(this);
```

- To declare indexers, for example:

```
C#

public int this[int param]
{
    get => array[param];
```

```
        set => array[param] = value;
    }
```

Static member functions, because they exist at the class level and not as part of an object, don't have a `this` pointer. It's an error to refer to `this` in a static method.

In this example, the parameters `name`, and `alias` hide fields with the same names. The `this` keyword qualifies those variables as `Employee` class members. The `this` keyword also specifies the object for the method `CalcTax`, which belongs to another class.

C#

```
class Employee
{
    private string name;
    private string alias;

    // Constructor:
    public Employee(string name, string alias)
    {
        // Use this to qualify the fields, name and alias:
        this.name = name;
        this.alias = alias;
    }

    // Printing method:
    public void printEmployee()
    {
        Console.WriteLine($"""
        Name: {name}
        Alias: {alias}
        """);
        // Passing the object to the CalcTax method by using this:
        Console.WriteLine($"Taxes: {Tax.CalcTax(this):C}");
    }

    public decimal Salary { get; } = 3000.00m;
}

class Tax
{
    public static decimal CalcTax(Employee E)=> 0.08m * E.Salary;
}

class Program
{
    static void Main()
    {
        // Create objects:
        Employee E1 = new Employee("Mingda Pan", "mpan");

        // Display results:
```

```
            E1.printEmployee();
        }
    }
    /*
    Output:
        Name: Mingda Pan
        Alias: mpan
        Taxes: $240.00
     */
```

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

- this code-style preferences (IDE0003 and IDE0009)
- C# Keywords
- base
- Methods

null (C# Reference)

Article • 03/30/2024

The `null` keyword is a literal that represents a null reference, one that does not refer to any object. `null` is the default value of reference-type variables. Ordinary value types cannot be null, except for nullable value types.

The following example demonstrates some behaviors of the `null` keyword:

C#

```
class Program
{
    class MyClass
    {
        public static void MyMethod() { }
    }

    static void Main()
    {
        // Set a breakpoint here to see that mc = null.
        // However, the compiler considers it "unassigned."
        // and generates a compiler error if you try to
        // use the variable.
        MyClass mc;

        // Now the variable can be used, but...
        mc = null;

        // ... a method call on a null object raises
        // a run-time NullReferenceException.
        // Uncomment the following line to see for yourself.
        // mc.MyMethod();

        // Now mc has a value.
        mc = new MyClass();

        // You can call its method.
        MyClass.MyMethod();

        // Set mc to null again. The object it referenced
        // is no longer accessible and can now be garbage-collected.
        mc = null;

        // A null string is not the same as an empty string.
        string s = null;
        string t = string.Empty; // Logically the same as ""

        // Equals applied to any null object returns false.
        Console.WriteLine($"t.Equals(s) is {t.Equals(s)}");
```

```
        // Equality operator also returns false when one
        // operand is null.
        Console.WriteLine($"Empty string {(s == t ? "equals" : "does not
equal")} null string");

        // Returns true.
        Console.WriteLine($"null == null is {null == null}");

        // A value type cannot be null
        // int i = null; // Compiler error!

        // Use a nullable value type instead:
        int? i = null;

        // Keep the console window open in debug mode.
    }
}
```

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

- C# keywords
- Default values of C# types
- Nothing (Visual Basic)

bool (C# reference)

Article • 01/25/2022

The `bool` type keyword is an alias for the .NET System.Boolean structure type that represents a Boolean value, which can be either `true` or `false`.

To perform logical operations with values of the `bool` type, use Boolean logical operators. The `bool` type is the result type of comparison and equality operators. A `bool` expression can be a controlling conditional expression in the if, do, while, and for statements and in the conditional operator ?:.

The default value of the `bool` type is `false`.

Literals

You can use the `true` and `false` literals to initialize a `bool` variable or to pass a `bool` value:

```
C#
```

```
bool check = true;
Console.WriteLine(check ? "Checked" : "Not checked");  // output: Checked

Console.WriteLine(false ? "Checked" : "Not checked");  // output: Not
checked
```

Three-valued Boolean logic

Use the nullable `bool?` type, if you need to support the three-valued logic, for example, when you work with databases that support a three-valued Boolean type. For the `bool?` operands, the predefined `&` and `|` operators support the three-valued logic. For more information, see the Nullable Boolean logical operators section of the Boolean logical operators article.

For more information about nullable value types, see Nullable value types.

Conversions

C# provides only two conversions that involve the `bool` type. Those are an implicit conversion to the corresponding nullable `bool?` type and an explicit conversion from

the `bool?` type. However, .NET provides additional methods that you can use to convert to or from the `bool` type. For more information, see the Converting to and from Boolean values section of the System.Boolean API reference page.

C# language specification

For more information, see The bool type section of the C# language specification.

See also

- Value types
- true and false operators

default (C# reference)

Article • 03/30/2024

You can use the `default` keyword in the following contexts:

- To specify the default case in the switch statement.
- As the default operator or literal to produce the default value of a type.
- As the default type constraint on a generic method override or explicit interface implementation.

See also

- C# keywords

The `add` contextual keyword (C# Reference)

Article • 03/15/2025

The `add` contextual keyword is used to define a custom event accessor that is invoked when client code subscribes to your event. If you supply a custom `add` accessor, you must also supply a remove accessor.

The following example shows an event that has custom `add` and remove accessors. For the full example, see How to implement interface events.

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

You don't typically need to provide your own custom event accessors. The automatically generated accessors when you declare an event are sufficient for most scenarios. Beginning with C# 14, you can declare partial events. The implementing declaration of a partial event must declare the `add` and `remove` handlers.

See also

- Events

Extension declaration (C# Reference)

09/17/2025

Beginning with C# 14, top level, nongeneric `static class` declarations can use `extension` blocks to declare *extension members*. Extension members are methods or properties and can appear to be instance or static members. Earlier versions of C# enable *extension methods* by adding `this` as a modifier to the first parameter of a static method declared in a top-level, nongeneric static class.

The `extension` block specifies the type and receiver for extension members. You can declare methods, properties, or operators inside the `extension` declaration. The following example declares a single extension block that defines an instance extension method, an instance property, and a static operator method.

C#

```
public static class NumericSequences
{
    extension(IEnumerable<int> sequence)
    {
        public IEnumerable<int> AddValue(int operand)
        {
            foreach (var item in sequence)
            {
                yield return item + operand;
            }
        }

        public int Median
        {
            get
            {

                var sortedList = sequence.OrderBy(n => n).ToList();
                int count = sortedList.Count;
                int middleIndex = count / 2;

                if (count % 2 == 0)
                {
                    // Even number of elements: average the two middle elements
                    return (sortedList[middleIndex - 1] +
sortedList[middleIndex]);
                }
                else
                {
                    // Odd number of elements: return the middle element
                    return sortedList[middleIndex];
                }
            }
```

```
        }

        public static IEnumerable<int> operator +(IEnumerable<int> left,
IEnumerable<int> right)
            => left.Concat(right);
    }
}
```

The `extension` defines the receiver: `sequence`, which is an `IEnumerable<int>`. The receiver type can be nongeneric, an open generic, or a closed generic type. The name `sequence` is in scope in every instance member declared in that extension. The extension method and property both access `sequence`.

Any of the extension members can be accessed as though they were members of the receiver type:

C#

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
numbers = numbers.AddValue(10);

var median = numbers.Median;

var combined = numbers + Enumerable.Range(100, 10);
```

You can declare any number of members in a single block, as long as they share the same receiver. You can declare as many extension blocks in a single class as well. Different extensions don't need to declare the same type or name of receiver. The extension parameter doesn't need to include the parameter name if the only members are static:

C#

```
extension(IEnumerable<int>)
{
    // Method:
    public static IEnumerable<int> Generate(int low, int count, int increment)
    {
        for (int i = 0; i < count; i++)
            yield return low + (i * increment);
    }

    // Property:
    public static IEnumerable<int> Identity => Enumerable.Empty<int>();
}
```

Static extensions can be called as though they're static members of the receiver type:

C#

```
var newSequence = IEnumerable<int>.Generate(5, 10, 2);
var identity = IEnumerable<int>.Identity;
```

Operators are called as though they're user defined operators on the type.

> **Important**
>
> An extension doesn't introduce a *scope* for member declarations. All members declared in a single class, even if in multiple extensions, must have unique signatures. The generated signature includes the receiver type in its name for static members and the receiver parameter for extension instance members.

The following example shows an extension method using the `this` modifier:

C#

```
public static class NumericSequenceExtensionMethods
{
    public static IEnumerable<int> AddValue(this IEnumerable<int> sequence, int
operand)
    {
        foreach (var item in sequence)
            yield return item + operand;
    }
}
```

The `Add` method can be called from any other method as though it was a member of the `IEnumerable<int>` interface:

C#

```
IEnumerable<int> numbers = Enumerable.Range(1, 10);
numbers = numbers.AddValue(10);
```

Both forms of extension methods generate the same intermediate language (IL). Callers can't make a distinction between them. In fact, you can convert existing extension methods to the new member syntax without a breaking change. The formats are both binary and source compatible.

Generic extension blocks

Where you specify the type parameters for an extension member declared in an extension block depends on where that type parameter is required:

- You add the type parameter to the `extension` declaration when the type parameter is used in the receiver.
- You add the type parameter to the member declaration when the type is distinct from any type parameter specified on the receiver.
- You can't specify the same type parameter in both locations.

The following example shows an extension block for `IEnumerable<T>` where two of the extension members require a second type parameter:

C#

```
public static class GenericExtensions
{
    extension<TReceiver>(IEnumerable<TReceiver> source)
    {
        public IEnumerable<TReceiver> Spread(int start, int count)
            => source.Skip(start).Take(count);

        public IEnumerable<TReceiver> Append<TArg>(IEnumerable<TArg> second,
Func<TArg, TReceiver> Converter)
        {
            foreach(TReceiver item in source)
            {
                yield return item;
            }
            foreach (TArg item in second)
            {
                yield return Converter(item);
            }
        }

        public IEnumerable<TReceiver> Prepend<TArg>(IEnumerable<TArg> second,
Func<TArg, TReceiver> Converter)
        {
            foreach (TArg item in second)
            {
                yield return Converter(item);
            }
            foreach (TReceiver item in source)
            {
                yield return item;
            }
        }
    }
}
```

The members `Append` and `Prepend` specify the *extra* type parameter for the conversion. None of the members repeat the type parameter for the receiver.

The equivalent extension method declarations demonstrate how those type parameters are encoded:

C#

```
public static class GenericExtensions
{
    public static IEnumerable<T> Spread<T>(this IEnumerable<T> source, int start,
int count)
        => source.Skip(start).Take(count);

    public static IEnumerable<T1> Append<T1, T2>(this IEnumerable<T1> source,
IEnumerable<T2> second, Func<T2, T1> Converter)
    {
        foreach (T1 item in source)
        {
            yield return item;
        }
        foreach (T2 item in second)
        {
            yield return Converter(item);
        }
    }

    public static IEnumerable<T1> Prepend<T1, T2>(this IEnumerable<T1> source,
IEnumerable<T2> second, Func<T2, T1> Converter)
    {
        foreach (T2 item in second)
        {
            yield return Converter(item);
        }
        foreach (T1 item in source)
        {
            yield return item;
        }
    }
}
```

See also

- Extensions feature specification

C# language specification

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

The `get` keyword

Article • 12/11/2024

The `get` keyword defines an *accessor* method in a property or indexer that returns the property value or the indexer element. For more information, see Properties, Automatically implemented Properties, and Indexers.

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can take advantage of the C# compiler's support for automatically implemented properties. The following example implements `Hours` as an automatically implemented property.

C#

```
class TimePeriod3
{
    public double Hours { get; set; }
}
```

> **ⓘ Important**
>
> Automatically implemented properties aren't allowed for interface property declarations or the implementing declaration for a partial property. The compiler interprets syntax matching an automatically implemented property as the declaring declaration, not an implementing declaration.

Often, the `get` accessor consists of a single statement that returns a value, as it did in the previous example. You can implement the `get` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessor as expression-bodied members.

C#

```
class TimePeriod2
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
```

```
    }
}
```

You might find that you need to implement one of the accessor bodies. You can use a field backed property to let the compiler generate one accessor while you write the other by hand. You use the `field` keyword, added as a preview feature in C# 13, to access the compiler synthesized backing field:

C#

```
class TimePeriod4
{
    public double Hours {
        get;
        set => field = (value >= 0)
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "The
value must not be negative");
    }
}
```

> **ⓘ Important**
>
> The `field` keyword is a preview feature in C# 13. You must be using .NET 9 and set your `<LangVersion>` element to `preview` in your project file in order to use the `field` contextual keyword.
>
> 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.

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
```

```
        set
        {
            if (value < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(value), "The
value of the time period must be non-negative.");
            }
            _seconds = value;
        }
    }
}
```

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

- C# Keywords
- Properties

The `init` keyword (C# Reference)

Article • 11/14/2024

The `init` keyword defines an *accessor* method in a property or indexer. An init-only setter assigns a value to the property or the indexer element **only** during object construction. An `init` enforces immutability, so that once the object is initialized, it can't be changed. An `init` accessor enables calling code to use an object initializer to set the initial value. As a contrast, an automatically implemented property with only a `get` setter must be initialized by calling a constructor. A property with a `private set` accessor can be modified after construction, but only in the class.

The following code demonstrates an `init` accessor in an automatically implemented property:

C#

```
class Person_InitExampleAutoProperty
{
    public int YearOfBirth { get; init; }
}
```

You might need to implement one of the accessors to provide parameter validation. You can do that using the `field` keyword, introduced as a preview feature in C# 13. The `field` keyword accesses the compiler synthesized backing field for that property. The following example shows a property where the `init` accessor validates the range of the `value` parameter"

C#

```
class Person_InitExampleFieldProperty
{
    public int YearOfBirth
    {
        get;
        init
        {
            field = (value <= DateTime.Now.Year)
                ? value
                : throw new ArgumentOutOfRangeException(nameof(value), "Year
of birth can't be in the future");
        }
    }
}
```

> **Important**
>
> The `field` keyword is a preview feature in C# 13. You must be using .NET 9 and set your `<LangVersion>` element to `preview` in your project file in order to use the `field` contextual keyword.
>
> 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.

The `init` accessor can be used as an expression-bodied member. Example:

C#

```
class Person_InitExampleExpressionBodied
{
    private int _yearOfBirth;

    public int YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

The following example defines both a `get` and an `init` accessor for a property named `YearOfBirth`. It uses a private field named `_yearOfBirth` to back the property value.

C#

```
class Person_InitExample
{
     private int _yearOfBirth;

     public int YearOfBirth
     {
         get { return _yearOfBirth; }
         init { _yearOfBirth = value; }
     }
}
```

An `init` accessor doesn't force callers to set the property. Instead, it allows callers to use an object initializer while prohibiting later modification. You can add the required

modifier to force callers to set a property. The following example shows an `init` only property with a nullable value type as its backing field. If a caller doesn't initialize the `YearOfBirth` property, that property has the default `null` value:

C#

```
class Person_InitExampleNullability
{
    private int? _yearOfBirth;

    public int? YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

To force callers to set an initial non-null value, you add the `required` modifier, as shown in the following example:

C#

```
class Person_InitExampleNonNull
{
    private int _yearOfBirth;

    public required int YearOfBirth
    {
        get => _yearOfBirth;
        init => _yearOfBirth = value;
    }
}
```

The following example shows the distinction between a `private set`, read only, and `init` property. Both the private set version and the read only version require callers to use the added constructor to set the name property. The `private set` version allows a person to change their name after the instance is constructed. The `init` version doesn't require a constructor. Callers can initialize the properties using an object initializer:

C#

```
class PersonPrivateSet
{
    public string FirstName { get; private set; }
    public string LastName { get; private set; }
    public PersonPrivateSet(string first, string last) => (FirstName,
LastName) = (first, last);
```

```
    public void ChangeName(string first, string last) => (FirstName,
LastName) = (first, last);
}

class PersonReadOnly
{
    public string FirstName { get; }
    public string LastName { get; }
    public PersonReadOnly(string first, string last) => (FirstName,
LastName) = (first, last);
}

class PersonInit
{
    public string FirstName { get; init; }
    public string LastName { get; init; }
}
```

C#

```
PersonPrivateSet personPrivateSet = new("Bill", "Gates");
PersonReadOnly personReadOnly = new("Bill", "Gates");
PersonInit personInit = new() { FirstName = "Bill", LastName = "Gates" };
```

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

Partial type (C# Reference)

Article • 03/15/2025

Partial type definitions allow for the definition of a class, struct, interface, or record to be split into multiple definitions. These multiple definitions can be in different files in the same project. One type declaration contains only the signatures for partial members:

C#

```
partial class A
{
    int num = 0;
    void MethodA() { }
    partial void MethodC();
}
```

The other declaration contains the implementation of the partial members:

C#

```
partial class A
{
    void MethodB() { }
    partial void MethodC() { }
}
```

The declarations for a partial type can appear in either the same or multiple files. Typically, the two declarations are in different files. You split a class, struct, or interface type when you're working with large projects, with automatically generated code such as that provided by the Windows Forms Designer, or Source generators like RegEx. A partial type can contain partial members.

Beginning with C# 13, you can define partial properties and partial indexers. Beginning with C# 14, you can define partial instance constructors and partial events. Before C# 13, only methods could be defined as partial members.

Documentation comments can be provided on either the declaring declaration or the implementing declaration. When documentation comments are applied to both type declarations, the XML elements from each declaration are included in the output XML. See the article on partial members for the rules on partial member declarations.

You can apply attributes to either declaration. All attributes are both declarations, including duplicates, are combined in the compiled output.

For more information, see Partial Classes and Methods.

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

- Modifiers
- Introduction to Generics

Partial member (C# Reference)

Article • 03/15/2025

A partial member has one *declaring declaration* and often one *implementing declaration*. The *declaring declaration* doesn't include a body. The *implementing declaration* provides the body of the member. Partial members enable class designers to provide member hooks for tooling such as source generators to implement. Partial types and members provide a way for human developers to write part of a type while tools write other parts of the type. If the developer doesn't supply an optional implementing declaration, the compiler can remove the declaring declaration at compile time. The following conditions apply to partial members:

- Declarations must begin with the contextual keyword partial.
- Signatures in both parts of the partial type must match.

The `partial` keyword isn't allowed on static constructors, finalizers, or overloaded operators. Before C# 14, `partial` wasn't allowed on instance constructors or event declarations. Before C# 13, `partial` wasn't allowed on properties or indexers.

A partial method isn't required to have an implementing declaration in the following cases:

- It doesn't have any accessibility modifiers (including the default private).
- It returns void.
- It doesn't have any out parameters.
- It doesn't have any of the following modifiers virtual, override, sealed, new, or extern.

The following example shows a partial method that conforms to the preceding restrictions:

C#

```
partial class MyPartialClass
{
    // Declaring definition
    partial void OnSomethingHappened(string s);
}

// This part can be in a separate file.
partial class MyPartialClass
{
    // Comment out this method and the program
    // will still compile.
    partial void OnSomethingHappened(string s) =>
```

```
        Console.WriteLine($"Something happened: {s}");
}
```

Any member that doesn't conform to all those restrictions (for example, `public virtual partial void` method), must provide an implementation.

Partial properties, indexers, and events can't use auto-implemented syntax for the implementing declaration. The defining declaration uses the same syntax. The implementing declaration must include at least one implemented accessor. That accessor can be a field-backed property. The implementing declaration for a partial event must define the `add` and `remove` handlers.

Partial members can also be useful in combination with source generators. For example a regex could be defined using the following pattern:

C#

```
public partial class RegExSourceGenerator
{
    [GeneratedRegex("cat|dog", RegexOptions.IgnoreCase, "en-US")]
    private static partial Regex CatOrDogGeneratedRegex();

    private static void EvaluateText(string text)
    {
        if (CatOrDogGeneratedRegex().IsMatch(text))
        {
            // Take action with matching text
        }
    }
}
```

The preceding example shows a partial method that must have an implementing declaration. As part of a build, the Regular expression source generator creates the implementing declaration.

The following example shows a declaring declaration and an implementing declaration for a class. Because the method's return type isn't `void` (it's `string`) and its access is `public`, the method must have an implementing declaration:

C#

```
// Declaring declaration
public partial class PartialExamples
{
    /// <summary>
    /// Gets or sets the number of elements that the List can contain.
    /// </summary>
```

```
    public partial int Capacity { get; set; }

    /// <summary>
    /// Gets or sets the element at the specified index.
    /// </summary>
    /// <param name="index">The index</param>
    /// <returns>The string stored at that index</returns>
    public partial string this[int index] { get; set; }

    public partial string? TryGetAt(int index);
}

public partial class PartialExamples
{
    private List<string> _items = [
        "one",
        "two",
        "three",
        "four",
        "five"
        ];

    // Implementing declaration

    /// <summary>
    /// Gets or sets the number of elements that the List can contain.
    /// </summary>
    /// <remarks>
    /// If the value is less than the current capacity, the list will shrink
to the
    /// new value. If the value is negative, the list isn't modified.
    /// </remarks>
    public partial int Capacity
    {
        get => _items.Count;
        set
        {
            if ((value != _items.Count) && (value >= 0))
            {
                _items.Capacity = value;
            }
        }
    }

    public partial string this[int index]
    {
        get => _items[index];
        set => _items[index] = value;
    }

    /// <summary>
    /// Gets the element at the specified index.
    /// </summary>
    /// <param name="index">The index</param>
    /// <returns>The string stored at that index, or null if out of
```

```
bounds</returns>
    public partial string? TryGetAt(int index)
    {
        if (index < _items.Count)
        {
            return _items[index];
        }
        return null;
    }
}
```

The preceding example illustrates rules on how the two declarations are combined:

- *Signature matches*: In general, the signatures for the declaring and implementing declarations must match. This includes accessibility modifier on methods, properties, indexers, and individual accessors. It includes the parameter type and ref-kind modifiers on all parameters. The return type and any ref-kind modifier must match. Tuple member names must match. However, some rules are flexible:
  - The declaring and implementing declarations can have different nullable annotations settings. Meaning that one can be *nullable oblivious* and the other *nullable enabled*.
  - Nullability differences that don't involve *oblivious nullability* generates a warning.
  - Default parameter values don't need to match. The compiler issues a warning if the implementing declaration of a method or indexer declares a default parameter value.
  - The compiler issues a warning when parameter names don't match. The emitted IL contains the declaring declaration's parameter names.
- *Documentation comments*: Documentation comments can be included from either declaration. If both the declaring and implementing declarations include documentation comments, the comments from the implementing declaration are included. In the preceding example, the documentation comments include:
  - For the `Capacity` property, the comments are taken from the implementing declaration. The implementing declaration comments are used when both declarations have `///` comments.
  - For the indexer, the comments are taken from the declaring declaration. The implementing declaration doesn't include any `///` comments.
  - For `TryGetAt`, the comments are taken from the implementing declaration. The declaring declaration doesn't include any `///` comments.
  - The generated XML has documentation comments for all `public` members.
- Most *Attribute declarations* are combined. However, all caller info attributes are defined with `AllowMultiple=false`. The compiler recognizes any caller info attribute on the declaring declaration. All caller info attributes on the

implementing declaration are ignored. The compiler issues a warning if you add caller info attributes on the implementing declaration.

See also

- partial type

The `remove` contextual keyword (C# Reference)

Article • 03/15/2025

The `remove` contextual keyword is used to define a custom event accessor that is invoked when client code unsubscribes from your event. If you supply a custom `remove` accessor, you must also supply an add accessor.

The following example shows an event with custom add and `remove` accessors. For the full example, see How to implement interface events.

C#

```
class Events : IDrawingObject
{
    event EventHandler PreDrawEvent;

    event EventHandler IDrawingObject.OnDraw
    {
        add => PreDrawEvent += value;
        remove => PreDrawEvent -= value;
    }
}
```

You don't typically need to provide your own custom event accessors. The automatically generated accessors when you declare an event are sufficient for most scenarios. Beginning with C# 14, you can declare partial events. The implementing declaration of a partial event must declare the `add` and `remove` handlers.

See also

- Events

required modifier (C# Reference)

Article • 01/31/2023

The `required` modifier indicates that the *field* or *property* it's applied to must be initialized by an object initializer. Any expression that initializes a new instance of the type must initialize all *required members*. The `required` modifier is available beginning with C# 11. The `required` modifier enables developers to create types where properties or fields must be properly initialized, yet still allow initialization using object initializers. Several rules ensure this behavior:

- The `required` modifier can be applied to *fields* and *properties* declared in `struct`, and `class` types, including `record` and `record struct` types. The `required` modifier can't be applied to members of an `interface`.
- Explicit interface implementations can't be marked as `required`. They can't be set in object initializers.
- Required members must be initialized, but they may be initialized to `null`. If the type is a non-nullable reference type, the compiler issues a warning if you initialize the member to `null`. The compiler issues an error if the member isn't initialized at all.
- Required members must be at least as visible as their containing type. For example, a `public` class can't contain a `required` field that's `protected`. Furthermore, required properties must have setters (`set` or `init` accessors) that are at least as visible as their containing types. Members that aren't accessible can't be set by code that creates an instance.
- Derived classes can't hide a `required` member declared in the base class. Hiding a required member prevents callers from using object initializers for it. Furthermore, derived types that override a required property must include the `required` modifier. The derived type can't remove the `required` state. Derived types can add the `required` modifier when overriding a property.
- A type with any `required` members may not be used as a type argument when the type parameter includes the `new()` constraint. The compiler can't enforce that all required members are initialized in the generic code.
- The `required` modifier isn't allowed on the declaration for positional parameters on a record. You can add an explicit declaration for a positional property that does include the `required` modifier.

Some types, such as positional records, use a primary constructor to initialize positional properties. If any of those properties include the `required` modifier, the primary constructor adds the SetsRequiredMembers attribute. This indicates that the primary

constructor initializes all required members. You can write your own constructor with the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute attribute. However, the compiler doesn't verify that these constructors do initialize all required members. Rather, the attribute asserts to the compiler that the constructor does initialize all required members. The `SetsRequiredMembers` attribute adds these rules to constructors:

- A constructor that chains to another constructor annotated with the `SetsRequiredMembers` attribute, either `this()`, or `base()`, must also include the `SetsRequiredMembers` attribute. That ensures that callers can correctly use all appropriate constructors.
- Copy constructors generated for `record` types have the `SetsRequiredMembers` attribute applied if any of the members are `required`.

> **⚠ Warning**
>
> The `SetsRequiredMembers` disables the compiler's checks that all `required` members are initialized when an object is created. Use it with caution.

The following code shows a class hierarchy that uses the `required` modifier for the `FirstName` and `LastName` properties:

C#

```
public class Person
{
    public Person() { }

    [SetsRequiredMembers]
    public Person(string firstName, string lastName) =>
        (FirstName, LastName) = (firstName, lastName);

    public required string FirstName { get; init; }
    public required string LastName { get; init; }

    public int? Age { get; set; }
}

public class Student : Person
{
    public Student() : base()
    {
    }

    [SetsRequiredMembers]
    public Student(string firstName, string lastName) :
        base(firstName, lastName)
    {
```

```
    }

    public double GPA { get; set; }
}
```

For more information on required members, see the C#11 - Required members feature specification.

The set keyword (C# Reference)

Article • 11/14/2024

The `set` keyword defines an *accessor* method in a property or indexer that assigns a value to the property or the indexer element. For more information and examples, see Properties, Automatically implemented properties, and Indexers.

For simple cases in which a property's `get` and `set` accessors perform no other operation than setting or retrieving a value in a private backing field, you can use automatically implemented properties. The following example implements `Hours` as an automatically implemented property.

C#

```
class TimePeriod3
{
    public double Hours { get; set; }
}
```

> **Important**
>
> Automatically implemented properties aren't allowed for interface property declarations or the implementing declaration for a partial property. The compiler interprets syntax matching an automatically implemented property as the declaring declaration, not an implementing declaration.

You might find that you need to implement one of the accessor bodies. The `field` keyword, added as a preview feature in C# 13 declares a field backed property. You can use a field backed property to let the compiler generate one accessor while you write the other by hand. You use the `field` keyword to access the compiler synthesized backing field:

C#

```
class TimePeriod4
{
    public double Hours {
        get;
        set => field = (value >= 0)
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "The
value must not be negative");
```

```
    }
}
```

> **Important**
>
> The `field` keyword is a preview feature in C# 13. You must be using .NET 9 and set your `<LangVersion>` element to `preview` in your project file in order to use the `field` contextual keyword.
>
> 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.

Often, the `set` accessor consists of a single statement that assigns a value, as it did in the previous example. You can implement the `set` accessor as an expression-bodied member. The following example implements both the `get` and the `set` accessors as expression-bodied members.

C#

```
class TimePeriod2
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

The following example defines both a `get` and a `set` accessor for a property named `Seconds`. It uses a private field named `_seconds` to back the property value.

C#

```
class TimePeriod
{
    private double _seconds;

    public double Seconds
    {
        get { return _seconds; }
```

```
        set
        {
            if (value < 0)
            {
                throw new ArgumentOutOfRangeException(nameof(value), "The
value of the time period must be non-negative.");
            }
            _seconds = value;
        }
    }
}
```

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

when (C# reference)

08/19/2025

You use the `when` contextual keyword to specify a filter condition in the following contexts:

- In a catch clause of a try-catch or try-catch-finally statement.
- As a case guard in the switch statement.
- As a case guard in the switch expression.

`when` in a catch clause

The `when` keyword can be used in a catch clause to specify a condition that must be true for the handler for a specific exception to execute. Its syntax is:

C#

```
catch (ExceptionType [e]) when (expr)
```

where *expr* is an expression that evaluates to a Boolean value. If it returns `true`, the exception handler executes; if `false`, it does not.

Exception filters with the `when` keyword provide several advantages over traditional exception handling approaches, including better debugging support and performance benefits. For a detailed explanation of how exception filters preserve the call stack and improve debugging, see Exception filters vs. traditional exception handling.

The following example uses the `when` keyword to conditionally execute handlers for an HttpRequestException depending on the text of the exception message.

C#

```
using System;
using System.Net.Http;
using System.Threading.Tasks;

class Program
{
    static void Main()
    {
        Console.WriteLine(MakeRequest().Result);
    }

    public static async Task<string> MakeRequest()
    {
        var client = new HttpClient();
```

```
        var streamTask = client.GetStringAsync("https://localHost:10000");
        try
        {
            var responseText = await streamTask;
            return responseText;
        }
        catch (HttpRequestException e) when (e.Message.Contains("301"))
        {
            return "Site Moved";
        }
        catch (HttpRequestException e) when (e.Message.Contains("404"))
        {
            return "Page Not Found";
        }
        catch (HttpRequestException e)
        {
            return e.Message;
        }
    }
}
```

See also

- C# keywords

`field` - Field backed property declarations

Article • 11/14/2024

> **ⓘ Important**
>
> The `field` keyword is a preview feature in C# 13. You must be using .NET 9 and set your `<LangVersion>` element to `preview` in your project file in order to use the `field` contextual keyword.
>
> 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.

The contextual keyword `field`, added as a preview feature in C# 13, can be used in a property accessor to access the compiler synthesized backing field of a property. This syntax enables you to define the body of a `get` or `set` accessor and let the compiler generate the other accessor as it would in an automatically implemented property.

The addition of the `field` contextual keywords provides a smooth path to add benefits such as range checking to an automatically implemented property. This practice is shown in the following example:

C#

```
class TimePeriod4
{
    public double Hours {
        get;
        set => field = (value >= 0)
            ? value
            : throw new ArgumentOutOfRangeException(nameof(value), "The
value must not be negative");
    }
}
```

You might implement the `Hours` property as an automatically implemented property. Then, you discover that you want to protect against a negative value. You use `field` and

provide range checking in the `set` accessor. You don't need to declare the backing field by hand and provide a body for the `get` accessor.

For more information, see the Properties and Indexers articles.

C# language specification

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

The `value` implicit parameter

Article • 01/01/2025

The implicit parameter `value` is used in the `set` accessor in property and indexer declarations. It's an input parameter of a method. The word `value` references the value that client code is attempting to assign to the property or indexer. In the following example, `TimePeriod2` has a property called `Seconds` that uses the `value` parameter to assign a new string to the backing field `_seconds`. From the point of view of client code, the operation is written as a simple assignment.

C#

```
class TimePeriod2
{
    private double _seconds;

    public double Seconds
    {
        get => _seconds;
        set => _seconds = value;
    }
}
```

For more information, see the Properties and Indexers articles.

C# language specification

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

Query keywords (C# Reference)

Article • 08/05/2023

This section contains the contextual keywords used in query expressions.

In this section

Expand table

Clause	Description
from	Specifies a data source and a range variable (similar to an iteration variable).
where	Filters source elements based on one or more Boolean expressions separated by logical AND and OR operators (`&&` or `||`).
select	Specifies the type and shape that the elements in the returned sequence will have when the query is executed.
group	Groups query results according to a specified key value.
into	Provides an identifier that can serve as a reference to the results of a join, group or select clause.
orderby	Sorts query results in ascending or descending order based on the default comparer for the element type.
join	Joins two data sources based on an equality comparison between two specified matching criteria.
let	Introduces a range variable to store subexpression results in a query expression.
in	Contextual keyword in a join clause.
on	Contextual keyword in a join clause.
equals	Contextual keyword in a join clause.
by	Contextual keyword in a group clause.
ascending	Contextual keyword in an orderby clause.
descending	Contextual keyword in an orderby clause.

See also

- C# Keywords

- LINQ in C#

from clause (C# Reference)

Article • 09/21/2022

A query expression must begin with a `from` clause. Additionally, a query expression can contain sub-queries, which also begin with a `from` clause. The `from` clause specifies the following:

- The data source on which the query or sub-query will be run.
- A local *range variable* that represents each element in the source sequence.

Both the range variable and the data source are strongly typed. The data source referenced in the `from` clause must have a type of IEnumerable, IEnumerable<T>, or a derived type such as IQueryable<T>.

In the following example, `numbers` is the data source and `num` is the range variable. Note that both variables are strongly typed even though the var keyword is used.

C#

```
class LowNums
{
    static void Main()
    {
        // A simple data source.
        int[] numbers = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];

        // Create the query.
        // lowNums is an IEnumerable<int>
        var lowNums = from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (int i in lowNums)
        {
            Console.Write(i + " ");
        }
    }
}
// Output: 4 1 3 2 0
```

The range variable

The compiler infers the type of the range variable when the data source implements IEnumerable<T>. For example, if the source has a type of `IEnumerable<Customer>`, then the range variable is inferred to be `Customer`. The only time that you must specify the type explicitly is when the source is a non-generic `IEnumerable` type such as ArrayList. For more information, see How to query an ArrayList with LINQ.

In the previous example `num` is inferred to be of type `int`. Because the range variable is strongly typed, you can call methods on it or use it in other operations. For example, instead of writing `select num`, you could write `select num.ToString()` to cause the query expression to return a sequence of strings instead of integers. Or you could write `select num + 10` to cause the expression to return the sequence 14, 11, 13, 12, 10. For more information, see select clause.

The range variable is like an iteration variable in a foreach statement except for one very important difference: a range variable never actually stores data from the source. It's just a syntactic convenience that enables the query to describe what will occur when the query is executed. For more information, see Introduction to LINQ Queries (C#).

Compound from clauses

In some cases, each element in the source sequence may itself be either a sequence or contain a sequence. For example, your data source may be an `IEnumerable<Student>` where each student object in the sequence contains a list of test scores. To access the inner list within each `Student` element, you can use compound `from` clauses. The technique is like using nested foreach statements. You can add where or orderby clauses to either `from` clause to filter the results. The following example shows a sequence of `Student` objects, each of which contains an inner `List` of integers representing test scores. To access the inner list, use a compound `from` clause. You can insert clauses between the two `from` clauses if necessary.

C#

```
class CompoundFrom
{
    // The element type of the data source.
    public class Student
    {
        public required string LastName { get; init; }
        public required List<int> Scores {get; init;}
    }

    static void Main()
    {
```

```
        // Use a collection initializer to create the data source. Note that
        // each element in the list contains an inner sequence of scores.
        List<Student> students =
        [
           new Student {LastName="Omelchenko", Scores= [97, 72, 81, 60]},
           new Student {LastName="O'Donnell", Scores= [75, 84, 91, 39]},
           new Student {LastName="Mortensen", Scores= [88, 94, 65, 85]},
           new Student {LastName="Garcia", Scores= [97, 89, 85, 82]},
           new Student {LastName="Beebe", Scores= [35, 72, 91, 70]}
        ];

        // Use a compound from to access the inner sequence within each
element.
        // Note the similarity to a nested foreach statement.
        var scoreQuery = from student in students
                         from score in student.Scores
                            where score > 90
                            select new { Last = student.LastName, score };

        // Execute the queries.
        Console.WriteLine("scoreQuery:");
        // Rest the mouse pointer on scoreQuery in the following line to
        // see its type. The type is IEnumerable<'a>, where 'a is an
        // anonymous type defined as new {string Last, int score}. That is,
        // each instance of this anonymous type has two members, a string
        // (Last) and an int (score).
        foreach (var student in scoreQuery)
        {
            Console.WriteLine($"{student.Last} Score: {student.score}");
        }
    }
}
/*
scoreQuery:
Omelchenko Score: 97
O'Donnell Score: 91
Mortensen Score: 94
Garcia Score: 97
Beebe Score: 91
*/
```

Using Multiple from Clauses to Perform Joins

A compound `from` clause is used to access inner collections in a single data source. However, a query can also contain multiple `from` clauses that generate supplemental queries from independent data sources. This technique enables you to perform certain types of join operations that are not possible by using the join clause.

The following example shows how two `from` clauses can be used to form a complete cross join of two data sources.

C#

```
class CompoundFrom2
{
    static void Main()
    {
        char[] upperCase = ['A', 'B', 'C'];
        char[] lowerCase = ['x', 'y', 'z'];

        // The type of joinQuery1 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery1 =
            from upper in upperCase
            from lower in lowerCase
            select new { upper, lower };

        // The type of joinQuery2 is IEnumerable<'a>, where 'a
        // indicates an anonymous type. This anonymous type has two
        // members, upper and lower, both of type char.
        var joinQuery2 =
            from lower in lowerCase
            where lower != 'x'
            from upper in upperCase
            select new { lower, upper };

        // Execute the queries.
        Console.WriteLine("Cross join:");
        // Rest the mouse pointer on joinQuery1 to verify its type.
        foreach (var pair in joinQuery1)
        {
            Console.WriteLine($"{pair.upper} is matched to {pair.lower}");
        }

        Console.WriteLine("Filtered non-equijoin:");
        // Rest the mouse pointer over joinQuery2 to verify its type.
        foreach (var pair in joinQuery2)
        {
            Console.WriteLine($"{pair.lower} is matched to {pair.upper}");
        }

        // Keep the console window open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
/* Output:
        Cross join:
        A is matched to x
        A is matched to y
        A is matched to z
        B is matched to x
        B is matched to y
        B is matched to z
```

```
        C is matched to x
        C is matched to y
        C is matched to z
        Filtered non-equijoin:
        y is matched to A
        y is matched to B
        y is matched to C
        z is matched to A
        z is matched to B
        z is matched to C
        */
```

For more information about join operations that use multiple `from` clauses, see Perform left outer joins.

See also

- Query Keywords (LINQ)
- Language Integrated Query (LINQ)

where clause (C# Reference)

Article • 09/15/2021

The `where` clause is used in a query expression to specify which elements from the data source will be returned in the query expression. It applies a Boolean condition (*predicate*) to each source element (referenced by the range variable) and returns those for which the specified condition is true. A single query expression may contain multiple `where` clauses and a single clause may contain multiple predicate subexpressions.

Example 1

In the following example, the `where` clause filters out all numbers except those that are less than five. If you remove the `where` clause, all numbers from the data source would be returned. The expression `num < 5` is the predicate that is applied to each element.

C#

```
class WhereSample
{
    static void Main()
    {
        // Simple data source. Arrays support IEnumerable<T>.
        int[] numbers = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];

        // Simple query with one predicate in where clause.
        var queryLowNums =
            from num in numbers
            where num < 5
            select num;

        // Execute the query.
        foreach (var s in queryLowNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }
}
//Output: 4 1 3 2 0
```

Example 2

Within a single `where` clause, you can specify as many predicates as necessary by using the && and || operators. In the following example, the query specifies two predicates in order to select only the even numbers that are less than five.

C#

```
class WhereSample2
{
static void Main()
{
    // Data source.
    int[] numbers = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];

    // Create the query with two predicates in where clause.
    var queryLowNums2 =
        from num in numbers
        where num < 5 && num % 2 == 0
        select num;

    // Execute the query
    foreach (var s in queryLowNums2)
    {
        Console.Write(s.ToString() + " ");
    }
    Console.WriteLine();

    // Create the query with two where clause.
    var queryLowNums3 =
        from num in numbers
        where num < 5
        where num % 2 == 0
        select num;

    // Execute the query
    foreach (var s in queryLowNums3)
    {
        Console.Write(s.ToString() + " ");
    }
}
}
// Output:
// 4 2 0
// 4 2 0
```

Example 3

A `where` clause may contain one or more methods that return Boolean values. In the following example, the `where` clause uses a method to determine whether the current value of the range variable is even or odd.

C#

```
class WhereSample3
{
```

```
    static void Main()
    {
        // Data source
        int[] numbers = [5, 4, 1, 3, 9, 8, 6, 7, 2, 0];

        // Create the query with a method call in the where clause.
        // Note: This won't work in LINQ to SQL unless you have a
        // stored procedure that is mapped to a method by this name.
        var queryEvenNums =
            from num in numbers
            where IsEven(num)
            select num;

         // Execute the query.
        foreach (var s in queryEvenNums)
        {
            Console.Write(s.ToString() + " ");
        }
    }

    // Method may be instance method or static method.
    static bool IsEven(int i) => i % 2 == 0;
}
//Output: 4 8 6 2 0
```

Remarks

The `where` clause is a filtering mechanism. It can be positioned almost anywhere in a query expression, except it cannot be the first or last clause. A `where` clause may appear either before or after a group clause depending on whether you have to filter the source elements before or after they are grouped.

If a specified predicate is not valid for the elements in the data source, a compile-time error will result. This is one benefit of the strong type-checking provided by LINQ.

At compile time the `where` keyword is converted into a call to the Where Standard Query Operator method.

See also

- Query Keywords (LINQ)
- from clause
- select clause
- Filtering Data
- LINQ in C#

select clause (C# Reference)

Article • 03/30/2024

In a query expression, the `select` clause specifies the type of values that will be produced when the query is executed. The result is based on the evaluation of all the previous clauses and on any expressions in the `select` clause itself. A query expression must terminate with either a `select` clause or a group clause.

The following example shows a simple `select` clause in a query expression.

C#

```
class SelectSample1
{
    static void Main()
    {
        //Create the data source
        List<int> Scores = [97, 92, 81, 60];

        // Create the query.
        IEnumerable<int> queryHighScores =
            from score in Scores
            where score > 80
            select score;

        // Execute the query.
        foreach (int i in queryHighScores)
        {
            Console.Write(i + " ");
        }
    }
}
//Output: 97 92 81
```

The type of the sequence produced by the `select` clause determines the type of the query variable `queryHighScores`. In the simplest case, the `select` clause just specifies the range variable. This causes the returned sequence to contain elements of the same type as the data source. For more information, see Type Relationships in LINQ Query Operations. However, the `select` clause also provides a powerful mechanism for transforming (or *projecting*) source data into new types. For more information, see Data Transformations with LINQ (C#).

Example

The following example shows all the different forms that a `select` clause may take. In each query, note the relationship between the `select` clause and the type of the *query variable* (`studentQuery1`, `studentQuery2`, and so on).

C#

```
class SelectSample2
{
    // Define some classes
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public required int ID { get; init; }
        public required List<int> Scores;
        public ContactInfo? GetContactInfo(SelectSample2 app, int id)
        {
            ContactInfo? cInfo =
                (from ci in app.contactList
                where ci.ID == id
                select ci)
                .FirstOrDefault();

            return cInfo;
        }

        public override string ToString() => $"{First} {Last}:{ID}";
    }

    public class ContactInfo
    {
        public required int ID { get; init; }
        public required string Email { get; init; }
        public required string Phone { get; init; }
        public override string ToString() => $"{Email},{Phone}";
    }

    public class ScoreInfo
    {
        public double Average { get; init; }
        public int ID { get; init; }
    }

    // The primary data source
    List<Student> students =
    [
         new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
new List<int>() {97, 92, 81, 60}},
         new Student {First="Claire", Last="O'Donnell", ID=112, Scores= new
List<int>() {75, 84, 91, 39}},
         new Student {First="Sven", Last="Mortensen", ID=113, Scores= new
List<int>() {88, 94, 65, 91}},
         new Student {First="Cesar", Last="Garcia", ID=114, Scores= new
```

```
List<int>() {97, 89, 85, 82}},
    ];

    // Separate data source for contact info.
    List<ContactInfo> contactList =
    [
        new ContactInfo {ID=111, Email="SvetlanO@Contoso.com", Phone="206-
555-0108"},
        new ContactInfo {ID=112, Email="ClaireO@Contoso.com", Phone="206-
555-0298"},
        new ContactInfo {ID=113, Email="SvenMort@Contoso.com", Phone="206-
555-1130"},
        new ContactInfo {ID=114, Email="CesarGar@Contoso.com", Phone="206-
555-0521"}
    ];

    static void Main(string[] args)
    {
        SelectSample2 app = new SelectSample2();

        // Produce a filtered sequence of unmodified Students.
        IEnumerable<Student> studentQuery1 =
            from student in app.students
            where student.ID > 111
            select student;

        Console.WriteLine("Query1: select range_variable");
        foreach (Student s in studentQuery1)
        {
            Console.WriteLine(s.ToString());
        }

        // Produce a filtered sequence of elements that contain
        // only one property of each Student.
        IEnumerable<String> studentQuery2 =
            from student in app.students
            where student.ID > 111
            select student.Last;

        Console.WriteLine("\r\n studentQuery2: select
range_variable.Property");
        foreach (string s in studentQuery2)
        {
            Console.WriteLine(s);
        }

        // Produce a filtered sequence of objects created by
        // a method call on each Student.
        IEnumerable<ContactInfo> studentQuery3 =
            from student in app.students
            where student.ID > 111
            select student.GetContactInfo(app, student.ID);

        Console.WriteLine("\r\n studentQuery3: select
range_variable.Method");
```

```
        foreach (ContactInfo ci in studentQuery3)
        {
            Console.WriteLine(ci.ToString());
        }

        // Produce a filtered sequence of ints from
        // the internal array inside each Student.
        IEnumerable<int> studentQuery4 =
            from student in app.students
            where student.ID > 111
            select student.Scores[0];

        Console.WriteLine("\r\n studentQuery4: select
range_variable[index]");
        foreach (int i in studentQuery4)
        {
            Console.WriteLine($"First score = {i}");
        }

        // Produce a filtered sequence of doubles
        // that are the result of an expression.
        IEnumerable<double> studentQuery5 =
            from student in app.students
            where student.ID > 111
            select student.Scores[0] * 1.1;

        Console.WriteLine("\r\n studentQuery5: select expression");
        foreach (double d in studentQuery5)
        {
            Console.WriteLine($"Adjusted first score = {d}");
        }

        // Produce a filtered sequence of doubles that are
        // the result of a method call.
        IEnumerable<double> studentQuery6 =
            from student in app.students
            where student.ID > 111
            select student.Scores.Average();

        Console.WriteLine("\r\n studentQuery6: select expression2");
        foreach (double d in studentQuery6)
        {
            Console.WriteLine($"Average = {d}");
        }

        // Produce a filtered sequence of anonymous types
        // that contain only two properties from each Student.
        var studentQuery7 =
            from student in app.students
            where student.ID > 111
            select new { student.First, student.Last };

        Console.WriteLine("\r\n studentQuery7: select new anonymous type");
        foreach (var item in studentQuery7)
        {
```

```
            Console.WriteLine("{0}, {1}", item.Last, item.First);
        }

        // Produce a filtered sequence of named objects that contain
        // a method return value and a property from each Student.
        // Use named types if you need to pass the query variable
        // across a method boundary.
        IEnumerable<ScoreInfo> studentQuery8 =
            from student in app.students
            where student.ID > 111
            select new ScoreInfo
            {
                Average = student.Scores.Average(),
                ID = student.ID
            };

        Console.WriteLine("\r\n studentQuery8: select new named type");
        foreach (ScoreInfo si in studentQuery8)
        {
            Console.WriteLine("ID = {0}, Average = {1}", si.ID, si.Average);
        }

        // Produce a filtered sequence of students who appear on a contact
list
        // and whose average is greater than 85.
        IEnumerable<ContactInfo> studentQuery9 =
            from student in app.students
            where student.Scores.Average() > 85
            join ci in app.contactList on student.ID equals ci.ID
            select ci;

        Console.WriteLine("\r\n studentQuery9: select result of join
clause");
        foreach (ContactInfo ci in studentQuery9)
        {
            Console.WriteLine("ID = {0}, Email = {1}", ci.ID, ci.Email);
        }
    }
}
/* Output
    Query1: select range_variable
    Claire O'Donnell:112
    Sven Mortensen:113
    Cesar Garcia:114

    studentQuery2: select range_variable.Property
    O'Donnell
    Mortensen
    Garcia

    studentQuery3: select range_variable.Method
    ClaireO@Contoso.com,206-555-0298
    SvenMort@Contoso.com,206-555-1130
    CesarGar@Contoso.com,206-555-0521
```

```
    studentQuery4: select range_variable[index]
    First score = 75
    First score = 88
    First score = 97

    studentQuery5: select expression
    Adjusted first score = 82.5
    Adjusted first score = 96.8
    Adjusted first score = 106.7

    studentQuery6: select expression2
    Average = 72.25
    Average = 84.5
    Average = 88.25

    studentQuery7: select new anonymous type
    O'Donnell, Claire
    Mortensen, Sven
    Garcia, Cesar

    studentQuery8: select new named type
    ID = 112, Average = 72.25
    ID = 113, Average = 84.5
    ID = 114, Average = 88.25

    studentQuery9: select result of join clause
    ID = 114, Email = CesarGar@Contoso.com
*/
```

As shown in `studentQuery8` in the previous example, sometimes you might want the elements of the returned sequence to contain only a subset of the properties of the source elements. By keeping the returned sequence as small as possible you can reduce the memory requirements and increase the speed of the execution of the query. You can accomplish this by creating an anonymous type in the `select` clause and using an object initializer to initialize it with the appropriate properties from the source element. For an example of how to do this, see Object and Collection Initializers.

Remarks

At compile time, the `select` clause is translated to a method call to the Select standard query operator.

See also

- Query Keywords (LINQ)
- from clause
- partial (Method) (C# Reference)

- Anonymous Types
- LINQ in C#

group clause (C# Reference)

Article • 09/15/2021

The `group` clause returns a sequence of IGrouping<TKey,TElement> objects that contain zero or more items that match the key value for the group. For example, you can group a sequence of strings according to the first letter in each string. In this case, the first letter is the key and has a type char, and is stored in the `Key` property of each IGrouping<TKey,TElement> object. The compiler infers the type of the key.

You can end a query expression with a `group` clause, as shown in the following example:

C#

```
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery1 =
    from student in students
    group student by student.Last[0];
```

If you want to perform additional query operations on each group, you can specify a temporary identifier by using the into contextual keyword. When you use `into`, you must continue with the query, and eventually end it with either a `select` statement or another `group` clause, as shown in the following excerpt:

C#

```
// Group students by the first letter of their last name
// Query variable is an IEnumerable<IGrouping<char, Student>>
var studentQuery2 =
    from student in students
    group student by student.Last[0] into g
    orderby g.Key
    select g;
```

More complete examples of the use of `group` with and without `into` are provided in the Example section of this article.

Enumerating the results of a group query

Because the IGrouping<TKey,TElement> objects produced by a `group` query are essentially a list of lists, you must use a nested foreach loop to access the items in each group. The outer loop iterates over the group keys, and the inner loop iterates over each

item in the group itself. A group may have a key but no elements. The following is the `foreach` loop that executes the query in the previous code examples:

C#

```
// Iterate group items with a nested foreach. This IGrouping encapsulates
// a sequence of Student objects, and a Key of type char.
// For convenience, var can also be used in the foreach statement.
foreach (IGrouping<char, Student> studentGroup in studentQuery2)
{
     Console.WriteLine(studentGroup.Key);
     // Explicit type for student could also be used here.
     foreach (var student in studentGroup)
     {
         Console.WriteLine("   {0}, {1}", student.Last, student.First);
     }
 }
```

Key types

Group keys can be any type, such as a string, a built-in numeric type, or a user-defined named type or anonymous type.

Grouping by string

The previous code examples used a `char`. A string key could easily have been specified instead, for example the complete last name:

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;
```

Grouping by bool

The following example shows the use of a bool value for a key to divide the results into two groups. Note that the value is produced by a sub-expression in the `group` clause.

C#

```
class GroupSample1
{
```

```
    // The element type of the data source.
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public required int ID { get; init; }
        public required List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students =
        [
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
[97, 72, 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= [99,
89, 91, 95]},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= [72,
81, 65, 84]},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= [97,
89, 85, 82]}
        ];

        return students;
    }

    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Group by true or false.
        // Query variable is an IEnumerable<IGrouping<bool, Student>>
        var booleanGroupQuery =
            from student in students
            group student by student.Scores.Average() >= 80; //pass or fail!

        // Execute the query and access items in each group
        foreach (var studentGroup in booleanGroupQuery)
        {
            Console.WriteLine(studentGroup.Key == true ? "High averages" :
"Low averages");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
            }
        }
    }
}
```

```
/* Output:
  Low averages
   Omelchenko, Svetlana:77.5
   O'Donnell, Claire:72.25
   Garcia, Cesar:75.5
  High averages
   Mortensen, Sven:93.5
   Garcia, Debra:88.25
*/
```

Grouping by numeric range

The next example uses an expression to create numeric group keys that represent a percentile range. Note the use of let as a convenient location to store a method call result, so that you don't have to call the method two times in the `group` clause. For more information about how to safely use methods in query expressions, see Handle exceptions in query expressions.

C#

```
class GroupSample2
{
    // The element type of the data source.
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public required int ID { get; init; }
        public required List<int> Scores;
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students =
        [
           new Student {First="Svetlana", Last="Omelchenko", ID=111, Scores=
[97, 72, 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= [99,
89, 91, 95]},
           new Student {First="Cesar", Last="Garcia", ID=114, Scores= [72,
81, 65, 84]},
           new Student {First="Debra", Last="Garcia", ID=115, Scores= [97,
89, 85, 82]}
        ];
```

```
        return students;
    }

    // This method groups students into percentile ranges based on their
    // grade average. The Average method returns a double, so to produce a
whole
    // number it is necessary to cast to int before dividing by 10.
    static void Main()
    {
        // Obtain the data source.
        List<Student> students = GetStudents();

        // Write the query.
        var studentQuery =
            from student in students
            let avg = (int)student.Scores.Average()
            group student by (avg / 10) into g
            orderby g.Key
            select g;

        // Execute the query.
        foreach (var studentGroup in studentQuery)
        {
            int temp = studentGroup.Key * 10;
            Console.WriteLine($"Students with an average between {temp} and
{temp + 10}");
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}:{2}", student.Last,
student.First, student.Scores.Average());
            }
        }
    }
}
/* Output:
     Students with an average between 70 and 80
       Omelchenko, Svetlana:77.5
       O'Donnell, Claire:72.25
       Garcia, Cesar:75.5
     Students with an average between 80 and 90
       Garcia, Debra:88.25
     Students with an average between 90 and 100
       Mortensen, Sven:93.5
 */
```

Grouping by composite keys

Use a composite key when you want to group elements according to more than one key. You create a composite key by using an anonymous type or a named type to hold the key element. In the following example, assume that a class `Person` has been

declared with members named `surname` and `city`. The `group` clause causes a separate group to be created for each set of persons with the same last name and the same city.

C#

```
group person by new {name = person.surname, city = person.city};
```

Use a named type if you must pass the query variable to another method. Create a special class using automatically implemented properties for the keys, and then override the Equals and GetHashCode methods. You can also use a struct, in which case you do not strictly have to override those methods. For more information see How to implement a lightweight class with automatically implemented properties and How to query for duplicate files in a directory tree. The latter article has a code example that demonstrates how to use a composite key with a named type.

Example 1

The following example shows the standard pattern for ordering source data into groups when no additional query logic is applied to the groups. This is called a grouping without a continuation. The elements in an array of strings are grouped according to their first letter. The result of the query is an IGrouping<TKey,TElement> type that contains a public `Key` property of type `char` and an IEnumerable<T> collection that contains each item in the grouping.

The result of a `group` clause is a sequence of sequences. Therefore, to access the individual elements within each returned group, use a nested `foreach` loop inside the loop that iterates the group keys, as shown in the following example.

C#

```
class GroupExample1
{
    static void Main()
    {
        // Create a data source.
        string[] words = ["blueberry", "chimpanzee", "abacus", "banana",
"apple", "cheese"];

        // Create the query.
        var wordGroups =
            from w in words
            group w by w[0];

        // Execute the query.
        foreach (var wordGroup in wordGroups)
```

```
        {
            Console.WriteLine($"Words that start with the letter
'{wordGroup.Key}':");
            foreach (var word in wordGroup)
            {
                Console.WriteLine(word);
            }
        }
    }
}
/* Output:
      Words that start with the letter 'b':
        blueberry
        banana
      Words that start with the letter 'c':
        chimpanzee
        cheese
      Words that start with the letter 'a':
        abacus
        apple
     */
```

Example 2

This example shows how to perform additional logic on the groups after you have created them, by using a *continuation* with `into`. For more information, see into. The following example queries each group to select only those whose key value is a vowel.

C#

```
class GroupClauseExample2
{
    static void Main()
    {
        // Create the data source.
        string[] words2 = ["blueberry", "chimpanzee", "abacus", "banana",
"apple", "cheese", "elephant", "umbrella", "anteater"];

        // Create the query.
        var wordGroups2 =
            from w in words2
            group w by w[0] into grps
            where (grps.Key == 'a' || grps.Key == 'e' || grps.Key == 'i'
                   || grps.Key == 'o' || grps.Key == 'u')
            select grps;

        // Execute the query.
        foreach (var wordGroup in wordGroups2)
        {
            Console.WriteLine($"Groups that start with a vowel:
{wordGroup.Key}");
```

```
            foreach (var word in wordGroup)
            {
                Console.WriteLine($"   {word}");
            }
        }
    }
}
/* Output:
    Groups that start with a vowel: a
        abacus
        apple
        anteater
    Groups that start with a vowel: e
        elephant
    Groups that start with a vowel: u
        umbrella
*/
```

Remarks

At compile time, `group` clauses are translated into calls to the GroupBy method.

Custom equality comparer is not supported in the syntax of `group` clause query. Use GroupBy method explicitly if you want to utilize IEqualityComparer in your query.

See also

- IGrouping<TKey,TElement>
- GroupBy
- ThenBy
- ThenByDescending
- Query Keywords
- Language Integrated Query (LINQ)
- Create a nested group
- Group query results
- Perform a subquery on a grouping operation

into (C# Reference)

Article • 09/15/2021

The `into` contextual keyword can be used to create a temporary identifier to store the results of a group, join or select clause into a new identifier. This identifier can itself be a generator for additional query commands. When used in a `group` or `select` clause, the use of the new identifier is sometimes referred to as a *continuation*.

Example

The following example shows the use of the `into` keyword to enable a temporary identifier `fruitGroup` which has an inferred type of `IGrouping`. By using the identifier, you can invoke the Count method on each group and select only those groups that contain two or more words.

C#

```
class IntoSample1
{
    static void Main()
    {

        // Create a data source.
        string[] words = ["apples", "blueberries", "oranges", "bananas",
"apricots"];

        // Create the query.
        var wordGroups1 =
            from w in words
            group w by w[0] into fruitGroup
            where fruitGroup.Count() >= 2
            select new { FirstLetter = fruitGroup.Key, Words =
fruitGroup.Count() };

        // Execute the query. Note that we only iterate over the groups,
        // not the items in each group
        foreach (var item in wordGroups1)
        {
            Console.WriteLine($" {item.FirstLetter} has {item.Words}
elements.");
        }
    }
}
/* Output:
   a has 2 elements.
```

```
    b has 2 elements.
*/
```

The use of `into` in a `group` clause is only necessary when you want to perform additional query operations on each group. For more information, see group clause.

For an example of the use of `into` in a `join` clause, see join clause.

See also

- Query Keywords (LINQ)
- LINQ in C#
- group clause

orderby clause (C# Reference)

Article • 03/30/2024

In a query expression, the `orderby` clause causes the returned sequence or subsequence (group) to be sorted in either ascending or descending order. Multiple keys can be specified in order to perform one or more secondary sort operations. The sorting is performed by the default comparer for the type of the element. The default sort order is ascending. You can also specify a custom comparer. However, it is only available by using method-based syntax. For more information, see Sorting Data.

Example 1

In the following example, the first query sorts the words in alphabetical order starting from A, and second query sorts the same words in descending order. (The `ascending` keyword is the default sort value and can be omitted.)

C#

```
class OrderbySample1
{
    static void Main()
    {
        // Create a delicious data source.
        string[] fruits = ["cherry", "apple", "blueberry"];

        // Query for ascending sort.
        IEnumerable<string> sortAscendingQuery =
            from fruit in fruits
            orderby fruit //"ascending" is default
            select fruit;

        // Query for descending sort.
        IEnumerable<string> sortDescendingQuery =
            from w in fruits
            orderby w descending
            select w;

        // Execute the query.
        Console.WriteLine("Ascending:");
        foreach (string s in sortAscendingQuery)
        {
            Console.WriteLine(s);
        }

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "Descending:");
        foreach (string s in sortDescendingQuery)
```

```
        {
            Console.WriteLine(s);
        }
    }
}
/* Output:
Ascending:
apple
blueberry
cherry

Descending:
cherry
blueberry
apple
*/
```

Example 2

The following example performs a primary sort on the students' last names, and then a secondary sort on their first names.

C#

```
class OrderbySample2
{
    // The element type of the data source.
    public class Student
    {
        public required string First { get; init; }
        public required string Last { get; init; }
        public int ID { get; set; }
    }

    public static List<Student> GetStudents()
    {
        // Use a collection initializer to create the data source. Note that
each element
        //  in the list contains an inner sequence of scores.
        List<Student> students = new List<Student>
        {
           new Student {First="Svetlana", Last="Omelchenko", ID=111},
           new Student {First="Claire", Last="O'Donnell", ID=112},
           new Student {First="Sven", Last="Mortensen", ID=113},
           new Student {First="Cesar", Last="Garcia", ID=114},
           new Student {First="Debra", Last="Garcia", ID=115}
        };

        return students;
    }
    static void Main(string[] args)
```

```
    {
        // Create the data source.
        List<Student> students = GetStudents();

        // Create the query.
        IEnumerable<Student> sortedStudents =
            from student in students
            orderby student.Last ascending, student.First ascending
            select student;

        // Execute the query.
        Console.WriteLine("sortedStudents:");
        foreach (Student student in sortedStudents)
            Console.WriteLine(student.Last + " " + student.First);

        // Now create groups and sort the groups. The query first sorts the
names
        // of all students so that they will be in alphabetical order after
they are
        // grouped. The second orderby sorts the group keys in alpha order.
        var sortedGroups =
            from student in students
            orderby student.Last, student.First
            group student by student.Last[0] into newGroup
            orderby newGroup.Key
            select newGroup;

        // Execute the query.
        Console.WriteLine(Environment.NewLine + "sortedGroups:");
        foreach (var studentGroup in sortedGroups)
        {
            Console.WriteLine(studentGroup.Key);
            foreach (var student in studentGroup)
            {
                Console.WriteLine("   {0}, {1}", student.Last,
student.First);
            }
        }
    }
}
/* Output:
sortedStudents:
Garcia Cesar
Garcia Debra
Mortensen Sven
O'Donnell Claire
Omelchenko Svetlana

sortedGroups:
G
   Garcia, Cesar
   Garcia, Debra
M
   Mortensen, Sven
O
```

```
    O'Donnell, Claire
    Omelchenko, Svetlana
*/
```

Remarks

At compile time, the `orderby` clause is translated to a call to the OrderBy method. Multiple keys in the `orderby` clause translate to ThenBy method calls.

See also

- Query Keywords (LINQ)
- LINQ in C#
- group clause
- Language Integrated Query (LINQ)

join clause (C# Reference)

Article • 04/28/2023

The `join` clause is useful for associating elements from different source sequences that have no direct relationship in the object model. The only requirement is that the elements in each source share some value that can be compared for equality. For example, a food distributor might have a list of suppliers of a certain product, and a list of buyers. A `join` clause can be used, for example, to create a list of the suppliers and buyers of that product who are all in the same specified region.

A `join` clause takes two source sequences as input. The elements in each sequence must either be or contain a property that can be compared to a corresponding property in the other sequence. The `join` clause compares the specified keys for equality by using the special `equals` keyword. All joins performed by the `join` clause are equijoins. The shape of the output of a `join` clause depends on the specific type of join you are performing. The following are three most common join types:

- Inner join
- Group join
- Left outer join

Inner join

The following example shows a simple inner equijoin. This query produces a flat sequence of "product name / category" pairs. The same category string will appear in multiple elements. If an element from `categories` has no matching `products`, that category will not appear in the results.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
//produces flat sequence
```

For more information, see Perform inner joins.

Group join

A `join` clause with an `into` expression is called a group join.

```
C#

var innerGroupJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    select new { CategoryName = category.Name, Products = prodGroup };
```

A group join produces a hierarchical result sequence, which associates elements in the left source sequence with one or more matching elements in the right side source sequence. A group join has no equivalent in relational terms; it is essentially a sequence of object arrays.

If no elements from the right source sequence are found to match an element in the left source, the `join` clause will produce an empty array for that item. Therefore, the group join is still basically an inner-equijoin except that the result sequence is organized into groups.

If you just select the results of a group join, you can access the items, but you cannot identify the key that they match on. Therefore, it is generally more useful to select the results of the group join into a new type that also has the key name, as shown in the previous example.

You can also, of course, use the result of a group join as the generator of another subquery:

```
C#

var innerGroupJoinQuery2 =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    from prod2 in prodGroup
    where prod2.UnitPrice > 2.50M
    select prod2;
```

For more information, see Perform grouped joins.

Left outer join

In a left outer join, all the elements in the left source sequence are returned, even if no matching elements are in the right sequence. To perform a left outer join in LINQ, use

the `DefaultIfEmpty` method in combination with a group join to specify a default right-side element to produce if a left-side element has no matches. You can use `null` as the default value for any reference type, or you can specify a user-defined default type. In the following example, a user-defined default type is shown:

```
C#
```

```
var leftOuterJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID into
prodGroup
    from item in prodGroup.DefaultIfEmpty(new Product { Name = String.Empty,
CategoryID = 0 })
    select new { CatName = category.Name, ProdName = item.Name };
```

For more information, see Perform left outer joins.

The equals operator

A `join` clause performs an equijoin. In other words, you can only base matches on the equality of two keys. Other types of comparisons such as "greater than" or "not equals" are not supported. To make clear that all joins are equijoins, the `join` clause uses the `equals` keyword instead of the `==` operator. The `equals` keyword can only be used in a `join` clause and it differs from the `==` operator in some important ways. When comparing strings, `equals` has an overload to compare by value and the operator `==` uses reference equality. When both sides of comparison have identical string variables, `equals` and `==` reach the same result: true. That's because, when a program declares two or more equivalent string variables, the compiler stores all of them in the same location. This is known as *interning*. Another important difference is the null comparison: `null equals null` is evaluated as false with `equals` operator, instead of `==` operator that evaluates it as true. Lastly, the scoping behavior is different: with `equals`, the left key consumes the outer source sequence, and the right key consumes the inner source. The outer source is only in scope on the left side of `equals` and the inner source sequence is only in scope on the right side.

Non-equijoins

You can perform non-equijoins, cross joins, and other custom join operations by using multiple `from` clauses to introduce new sequences independently into a query. For more information, see Perform custom join operations.

Joins on object collections vs. relational tables

In a LINQ query expression, join operations are performed on object collections. Object collections cannot be "joined" in exactly the same way as two relational tables. In LINQ, explicit `join` clauses are only required when two source sequences are not tied by any relationship. When working with LINQ to SQL, foreign key tables are represented in the object model as properties of the primary table. For example, in the Northwind database, the Customer table has a foreign key relationship with the Orders table. When you map the tables to the object model, the Customer class has an Orders property that contains the collection of Orders associated with that Customer. In effect, the join has already been done for you.

For more information about querying across related tables in the context of LINQ to SQL, see How to: Map Database Relationships.

Composite keys

You can test for equality of multiple values by using a composite key. For more information, see Join by using composite keys. Composite keys can be also used in a `group` clause.

Example

The following example compares the results of an inner join, a group join, and a left outer join on the same data sources by using the same matching keys. Some extra code is added to these examples to clarify the results in the console display.

C#

```
class JoinDemonstration
{
    #region Data

    class Product
    {
        public required string Name { get; init; }
        public required int CategoryID { get; init; }
    }

    class Category
    {
        public required string Name { get; init; }
        public required int ID { get; init; }
    }
```

```
    // Specify the first data source.
    List<Category> categories =
    [
        new Category {Name="Beverages", ID=001},
        new Category {Name="Condiments", ID=002},
        new Category {Name="Vegetables", ID=003},
        new Category {Name="Grains", ID=004},
        new Category {Name="Fruit", ID=005}
    ];

    // Specify the second data source.
    List<Product> products =
    [
      new Product {Name="Cola",  CategoryID=001},
      new Product {Name="Tea",  CategoryID=001},
      new Product {Name="Mustard", CategoryID=002},
      new Product {Name="Pickles", CategoryID=002},
      new Product {Name="Carrots", CategoryID=003},
      new Product {Name="Bok Choy", CategoryID=003},
      new Product {Name="Peaches", CategoryID=005},
      new Product {Name="Melons", CategoryID=005},
    ];
    #endregion

    static void Main(string[] args)
    {
        JoinDemonstration app = new JoinDemonstration();

        app.InnerJoin();
        app.GroupJoin();
        app.GroupInnerJoin();
        app.GroupJoin3();
        app.LeftOuterJoin();
        app.LeftOuterJoin2();
    }

    void InnerJoin()
    {
        // Create the query that selects
        // a property from each element.
        var innerJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID
           select new { Category = category.ID, Product = prod.Name };

        Console.WriteLine("InnerJoin:");
        // Execute the query. Access results
        // with a simple foreach statement.
        foreach (var item in innerJoinQuery)
        {
            Console.WriteLine("{0,-10}{1}", item.Product, item.Category);
        }
        Console.WriteLine($"InnerJoin: {innerJoinQuery.Count()} items in 1
group.");
        Console.WriteLine(System.Environment.NewLine);
```

```
    }

    void GroupJoin()
    {
        // This is a demonstration query to show the output
        // of a "raw" group join. A more typical group join
        // is shown in the GroupInnerJoin method.
        var groupJoinQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           select prodGroup;

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Simple GroupJoin:");

        // A nested foreach statement is required to access group items.
        foreach (var prodGrouping in groupJoinQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("   {0,-10}{1}", item.Name,
item.CategoryID);
            }
        }
        Console.WriteLine($"Unshaped GroupJoin: {totalItems} items in
{groupJoinQuery.Count()} unnamed groups");
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupInnerJoin()
    {
        var groupJoinQuery2 =
            from category in categories
            orderby category.ID
            join prod in products on category.ID equals prod.CategoryID into
prodGroup
            select new
            {
                Category = category.Name,
                Products = from prod2 in prodGroup
                           orderby prod2.Name
                           select prod2
            };

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupInnerJoin:");
        foreach (var productGroup in groupJoinQuery2)
        {
```

```
            Console.WriteLine(productGroup.Category);
            foreach (var prodItem in productGroup.Products)
            {
                totalItems++;
                Console.WriteLine("  {0,-10} {1}", prodItem.Name,
prodItem.CategoryID);
            }
        }
        Console.WriteLine($"GroupInnerJoin: {totalItems} items in
{groupJoinQuery2.Count()} named groups");
        Console.WriteLine(System.Environment.NewLine);
    }

    void GroupJoin3()
    {

        var groupJoinQuery3 =
            from category in categories
            join product in products on category.ID equals
product.CategoryID into prodGroup
            from prod in prodGroup
            orderby prod.CategoryID
            select new { Category = prod.CategoryID, ProductName = prod.Name
};

        //Console.WriteLine("GroupInnerJoin:");
        int totalItems = 0;

        Console.WriteLine("GroupJoin3:");
        foreach (var item in groupJoinQuery3)
        {
            totalItems++;
            Console.WriteLine($"   {item.ProductName}:{item.Category}");
        }

        Console.WriteLine($"GroupJoin3: {totalItems} items in 1 group");
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin()
    {
        // Create the query.
        var leftOuterQuery =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           select prodGroup.DefaultIfEmpty(new Product() { Name =
"Nothing!", CategoryID = category.ID });

        // Store the count of total items (for demonstration only).
        int totalItems = 0;

        Console.WriteLine("Left Outer Join:");

        // A nested foreach statement  is required to access group items
```

```
        foreach (var prodGrouping in leftOuterQuery)
        {
            Console.WriteLine("Group:");
            foreach (var item in prodGrouping)
            {
                totalItems++;
                Console.WriteLine("  {0,-10}{1}", item.Name,
item.CategoryID);
            }
        }
        Console.WriteLine($"LeftOuterJoin: {totalItems} items in
{leftOuterQuery.Count()} groups");
        Console.WriteLine(System.Environment.NewLine);
    }

    void LeftOuterJoin2()
    {
        // Create the query.
        var leftOuterQuery2 =
           from category in categories
           join prod in products on category.ID equals prod.CategoryID into
prodGroup
           from item in prodGroup.DefaultIfEmpty()
           select new { Name = item == null ? "Nothing!" : item.Name,
CategoryID = category.ID };

        Console.WriteLine($"LeftOuterJoin2: {leftOuterQuery2.Count()} items
in 1 group");
        // Store the count of total items
        int totalItems = 0;

        Console.WriteLine("Left Outer Join 2:");

        // Groups have been flattened.
        foreach (var item in leftOuterQuery2)
        {
            totalItems++;
            Console.WriteLine("{0,-10}{1}", item.Name, item.CategoryID);
        }
        Console.WriteLine($"LeftOuterJoin2: {totalItems} items in 1 group");
    }
}
/*Output:

InnerJoin:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Peaches   5
Melons    5
InnerJoin: 8 items in 1 group.
```

```
Unshaped GroupJoin:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
    Carrots   3
    Bok Choy  3
Group:
Group:
    Peaches   5
    Melons    5
Unshaped GroupJoin: 8 items in 5 unnamed groups


GroupInnerJoin:
Beverages
    Cola       1
    Tea        1
Condiments
    Mustard    2
    Pickles    2
Vegetables
    Bok Choy   3
    Carrots    3
Grains
Fruit
    Melons     5
    Peaches    5
GroupInnerJoin: 8 items in 5 named groups


GroupJoin3:
    Cola:1
    Tea:1
    Mustard:2
    Pickles:2
    Carrots:3
    Bok Choy:3
    Peaches:5
    Melons:5
GroupJoin3: 8 items in 1 group


Left Outer Join:
Group:
    Cola      1
    Tea       1
Group:
    Mustard   2
    Pickles   2
Group:
```

```
    Carrots   3
    Bok Choy  3
Group:
    Nothing!  4
Group:
    Peaches   5
    Melons    5
LeftOuterJoin: 9 items in 5 groups


LeftOuterJoin2: 9 items in 1 group
Left Outer Join 2:
Cola      1
Tea       1
Mustard   2
Pickles   2
Carrots   3
Bok Choy  3
Nothing!  4
Peaches   5
Melons    5
LeftOuterJoin2: 9 items in 1 group
Press any key to exit.
*/
```

Remarks

A `join` clause that is not followed by `into` is translated into a Join method call. A `join` clause that is followed by `into` is translated to a GroupJoin method call.

See also

- Query Keywords (LINQ)
- Language Integrated Query (LINQ)
- Join Operations
- group clause
- Perform left outer joins
- Perform inner joins
- Perform grouped joins
- Order the results of a join clause
- Join by using composite keys
- Compatible database systems for Visual Studio

let clause (C# Reference)

Article • 03/30/2024

In a query expression, it's sometimes useful to store the result of a subexpression in order to use it in subsequent clauses. You can do this with the `let` keyword, which creates a new range variable and initializes it with the result of the expression you supply. Once initialized with a value, the range variable can't be used to store another value. However, if the range variable holds a queryable type, it can be queried.

Example

In the following example `let` is used in two ways:

1. To create an enumerable type that can itself be queried.
2. To enable the query to call `ToLower` only one time on the range variable `word`. Without using `let`, you would have to call `ToLower` in each predicate in the `where` clause.

C#

```
class LetSample1
{
    static void Main()
    {
        string[] strings =
        [
            "A penny saved is a penny earned.",
            "The early bird catches the worm.",
            "The pen is mightier than the sword."
        ];

        // Split the sentence into an array of words
        // and select those whose first letter is a vowel.
        var earlyBirdQuery =
            from sentence in strings
            let words = sentence.Split(' ')
            from word in words
            let w = word.ToLower()
            where w[0] == 'a' || w[0] == 'e'
                || w[0] == 'i' || w[0] == 'o'
                || w[0] == 'u'
            select word;

        // Execute the query.
        foreach (var v in earlyBirdQuery)
        {
```

```
            Console.WriteLine($"\"{v}\" starts with a vowel");
        }
    }
}
/* Output:
    "A" starts with a vowel
    "is" starts with a vowel
    "a" starts with a vowel
    "earned." starts with a vowel
    "early" starts with a vowel
    "is" starts with a vowel
*/
```

See also

- Query Keywords (LINQ)
- LINQ in C#
- Language Integrated Query (LINQ)
- Handle exceptions in query expressions

ascending (C# Reference)

Article • 03/30/2024

The `ascending` contextual keyword is used in the orderby clause in query expressions to specify that the sort order is from smallest to largest. Because `ascending` is the default sort order, you do not have to specify it.

Example

The following example shows the use of `ascending` in an orderby clause.

C#

```
IEnumerable<string> sortAscendingQuery =
    from vegetable in vegetables
    orderby vegetable ascending
    select vegetable;
```

See also

- LINQ in C#
- descending

descending (C# Reference)

Article • 03/30/2024

The `descending` contextual keyword is used in the orderby clause in query expressions to specify that the sort order is from largest to smallest.

Example

The following example shows the use of `descending` in an orderby clause.

C#

```
IEnumerable<string> sortDescendingQuery =
    from vegetable in vegetables
    orderby vegetable descending
    select vegetable;
```

See also

- LINQ in C#
- ascending

on (C# Reference)

Article • 03/30/2024

The `on` contextual keyword is used in the join clause of a query expression to specify the join condition.

Example

The following example shows the use of `on` in a `join` clause.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- Language Integrated Query (LINQ)

equals (C# Reference)

Article • 09/15/2021

The `equals` contextual keyword is used in a `join` clause in a query expression to compare the elements of two sequences. For more information, see join clause.

Example

The following example shows the use of the `equals` keyword in a `join` clause.

C#

```
var innerJoinQuery =
    from category in categories
    join prod in products on category.ID equals prod.CategoryID
    select new { ProductName = prod.Name, Category = category.Name };
```

See also

- Language Integrated Query (LINQ)

by (C# Reference)

Article • 09/15/2021

The `by` contextual keyword is used in the `group` clause in a query expression to specify how the returned items should be grouped. For more information, see group clause.

Example

The following example shows the use of the `by` contextual keyword in a `group` clause to specify that the students should be grouped according to the first letter of the last name of each student.

C#

```
var query = from student in students
            group student by student.LastName[0];
```

See also

- LINQ in C#

in (C# Reference)

Article • 03/30/2024

The `in` keyword is used in the following contexts:

- generic type parameters in generic interfaces and delegates.
- As a parameter modifier, which lets you pass an argument to a method by reference rather than by value.
- foreach statements.
- from clauses in LINQ query expressions.
- join clauses in LINQ query expressions.

See also

- C# Keywords

C# operators and expressions

Article • 03/09/2023

C# provides a number of operators. Many of them are supported by the built-in types and allow you to perform basic operations with values of those types. Those operators include the following groups:

- Arithmetic operators that perform arithmetic operations with numeric operands
- Comparison operators that compare numeric operands
- Boolean logical operators that perform logical operations with bool operands
- Bitwise and shift operators that perform bitwise or shift operations with operands of the integral types
- Equality operators that check if their operands are equal or not

Typically, you can overload those operators, that is, specify the operator behavior for the operands of a user-defined type.

The simplest C# expressions are literals (for example, integer and real numbers) and names of variables. You can combine them into complex expressions by using operators. Operator precedence and associativity determine the order in which the operations in an expression are performed. You can use parentheses to change the order of evaluation imposed by operator precedence and associativity.

In the following code, examples of expressions are at the right-hand side of assignments:

C#

```
int a, b, c;
a = 7;
b = a;
c = b++;
b = a + b * c;
c = a >= 100 ? b : c / 10;
a = (int)Math.Sqrt(b * b + c * c);

string s = "String literal";
char l = s[s.Length - 1];

List<int> numbers = [..collection];
b = numbers.FindLast(n => n > 1);
```

Typically, an expression produces a result and can be included in another expression. A void method call is an example of an expression that doesn't produce a result. It can be used only as a statement, as the following example shows:

C#

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

Here are some other kinds of expressions that C# provides:

- Interpolated string expressions that provide convenient syntax to create formatted strings:

  C#

  ```
  var r = 2.3;
  var message = $"The area of a circle with radius {r} is {Math.PI * r * r:F3}.";
  Console.WriteLine(message);
  // Output:
  // The area of a circle with radius 2.3 is 16.619.
  ```

- Lambda expressions that allow you to create anonymous functions:

  C#

  ```
  int[] numbers = { 2, 3, 4, 5 };
  var maximumSquare = numbers.Max(x => x * x);
  Console.WriteLine(maximumSquare);
  // Output:
  // 25
  ```

- Query expressions that allow you to use query capabilities directly in C#:

  C#

  ```
  int[] scores = { 90, 97, 78, 68, 85 };
  IEnumerable<int> highScoresQuery =
      from score in scores
      where score > 80
      orderby score descending
      select score;
  Console.WriteLine(string.Join(" ", highScoresQuery));
  // Output:
  // 97 90 85
  ```

You can use an expression body definition to provide a concise definition for a method, constructor, property, indexer, or finalizer.

Operator precedence

In an expression with multiple operators, the operators with higher precedence are evaluated before the operators with lower precedence. In the following example, the multiplication is performed first because it has higher precedence than addition:

```
C#

var a = 2 + 2 * 2;
Console.WriteLine(a); //  output: 6
```

Use parentheses to change the order of evaluation imposed by operator precedence:

```
C#

var a = (2 + 2) * 2;
Console.WriteLine(a); //  output: 8
```

The following table lists the C# operators starting with the highest precedence to the lowest. The operators within each row have the same precedence.

Expand table

Operators	Category or name
x.y, f(x), a[i], x?.y, x?[y], x++, x--, x!, new, typeof, checked, unchecked, default, nameof, delegate, sizeof, stackalloc, x->y	Primary
+x, -x, !x, ~x, ++x, --x, ^x, (T)x, await, &x, *x, true and false	Unary
x..y	Range
switch, with	`switch` and `with` expressions
x * y, x / y, x % y	Multiplicative
x + y, x – y	Additive
x << y, x >> y, x >>> y	Shift
x < y, x > y, x <= y, x >= y, is, as	Relational and type-testing
x == y, x != y	Equality
`x & y`	Boolean logical AND or bitwise logical AND
`x ^ y`	Boolean logical XOR or bitwise logical XOR

Operators	Category or name
`x | y`	Boolean logical OR or bitwise logical OR
x && y	Conditional AND
x || y	Conditional OR
x ?? y	Null-coalescing operator
c ? t : f	Conditional operator
x = y, x += y, x -= y, x *= y, x /= y, x %= y, x &= y, x |= y, x ^= y, x <<= y, x >>= y, x >>>= y, x ??= y, =>	Assignment and lambda declaration

For information about the precedence of logical pattern combinators, see the Precedence and order of checking of logical patterns section of the Patterns article.

Operator associativity

When operators have the same precedence, associativity of the operators determines the order in which the operations are performed:

- *Left-associative* operators are evaluated in order from left to right. Except for the assignment operators and the null-coalescing operators, all binary operators are left-associative. For example, `a + b - c` is evaluated as `(a + b) - c`.
- *Right-associative* operators are evaluated in order from right to left. The assignment operators, the null-coalescing operators, lambdas, and the conditional operator ?: are right-associative. For example, `x = y = z` is evaluated as `x = (y = z)`.

> **Important**
>
> In an expression of the form `P?.A0?.A1`, if `P` is `null`, neither `A0` nor `A1` are evaluated. Similarly, in an expression of the form `P?.A0.A1`, because `A0` isn't evaluated when `P` is null, neither is `A0.A1`. See the C# language specification for more details.

Use parentheses to change the order of evaluation imposed by operator associativity:

```
C#
```

```
int a = 13 / 5 / 2;
int b = 13 / (5 / 2);
Console.WriteLine($"a = {a}, b = {b}");  // output: a = 1, b = 6
```

Operand evaluation

Unrelated to operator precedence and associativity, operands in an expression are evaluated from left to right. The following examples demonstrate the order in which operators and operands are evaluated:

Expand table

Expression	Order of evaluation
`a + b`	a, b, +
`a + b * c`	a, b, c, *, +
`a / b + c * d`	a, b, /, c, d, *, +
`a / (b + c) * d`	a, b, c, +, /, d, *

Typically, all operator operands are evaluated. However, some operators evaluate operands conditionally. That is, the value of the leftmost operand of such an operator defines if (or which) other operands should be evaluated. These operators are the conditional logical AND (&&) and OR (||) operators, the null-coalescing operators ?? and ??=, the null-conditional operators ?. and ?[], and the conditional operator ?:. For more information, see the description of each operator.

C# language specification

For more information, see the following sections of the C# language specification:

- Expressions
- Operators

See also

- Operator overloading
- Expression trees

Arithmetic operators (C# reference)

06/11/2025

The following operators perform arithmetic operations with operands of numeric types:

- Unary ++ (increment), -- (decrement), + (plus), and - (minus) operators
- Binary * (multiplication), / (division), % (remainder), + (addition), and - (subtraction) operators

All integral and floating-point numeric types support these operators.

The `int`, `uint`, `long`, and `ulong` types define all these operators. The other integral types (`sbyte`, `byte`, `short`, `ushort`, and `char`) define only the `++` and `--` operators. For other operators, when operands are of the integral types `sbyte`, `byte`, `short`, `ushort`, or `char`, their values are converted to the `int` type and the result type is `int`. When operands are of different integral or floating-point types, their values are converted to the closest containing type, if such a type exists. For more information, see the Numeric promotions section of the C# language specification. The `++` and `--` operators are defined for all integral and floating-point numeric types and the char type. The result type of a compound assignment expression is the type of the left-hand operand.

Increment operator ++

The unary increment operator `++` increments its operand by 1. The operand must be a variable, a property access, or an indexer access.

The increment operator is supported in two forms: the postfix increment operator, `x++`, and the prefix increment operator, `++x`.

Postfix increment operator

The result of `x++` is the value of `x` *before* the operation, as the following example shows:

C#

```
int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i++); // output: 3
Console.WriteLine(i);   // output: 4
```

Prefix increment operator

The result of `++x` is the value of `x` *after* the operation, as the following example shows:

```
C#

double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(++a); // output: 2.5
Console.WriteLine(a);   // output: 2.5
```

Decrement operator --

The unary decrement operator `--` decrements its operand by 1. The operand must be a variable, a property access, or an indexer access.

The decrement operator is supported in two forms: the postfix decrement operator, `x--`, and the prefix decrement operator, `--x`.

Postfix decrement operator

The result of `x--` is the value of `x` *before* the operation, as the following example shows:

```
C#

int i = 3;
Console.WriteLine(i);   // output: 3
Console.WriteLine(i--); // output: 3
Console.WriteLine(i);   // output: 2
```

Prefix decrement operator

The result of `--x` is the value of `x` *after* the operation, as the following example shows:

```
C#

double a = 1.5;
Console.WriteLine(a);   // output: 1.5
Console.WriteLine(--a); // output: 0.5
Console.WriteLine(a);   // output: 0.5
```

Unary plus and minus operators

The unary `+` operator returns the value of its operand. The unary `-` operator computes the numeric negation of its operand.

```
C#

Console.WriteLine(+4);     // output: 4

Console.WriteLine(-4);     // output: -4
Console.WriteLine(-(-4));  // output: 4

uint a = 5;
var b = -a;
Console.WriteLine(b);            // output: -5
Console.WriteLine(b.GetType());  // output: System.Int64

Console.WriteLine(-double.NaN);  // output: NaN
```

The ulong type doesn't support the unary `-` operator.

Multiplication operator *

The multiplication operator `*` computes the product of its operands:

```
C#

Console.WriteLine(5 * 2);         // output: 10
Console.WriteLine(0.5 * 2.5);     // output: 1.25
Console.WriteLine(0.1m * 23.4m);  // output: 2.34
```

The unary `*` operator is the pointer indirection operator.

Division operator /

The division operator `/` divides its left-hand operand by its right-hand operand.

Integer division

For the operands of integer types, the result of the `/` operator is of an integer type and equals the quotient of the two operands rounded towards zero:

```
C#

Console.WriteLine(13 / 5);    // output: 2
Console.WriteLine(-13 / 5);   // output: -2
```

```
Console.WriteLine(13 / -5);   // output: -2
Console.WriteLine(-13 / -5);  // output: 2
```

To obtain the quotient of the two operands as a floating-point number, use the `float`, `double`, or `decimal` type:

C#

```
Console.WriteLine(13 / 5.0);       // output: 2.6

int a = 13;
int b = 5;
Console.WriteLine((double)a / b);  // output: 2.6
```

Floating-point division

For the `float`, `double`, and `decimal` types, the result of the `/` operator is the quotient of the two operands:

C#

```
Console.WriteLine(16.8f / 4.1f);   // output: 4.097561
Console.WriteLine(16.8d / 4.1d);   // output: 4.09756097560976
Console.WriteLine(16.8m / 4.1m);   // output: 4.0975609756097560975609756098
```

If one of the operands is `decimal`, another operand can't be `float` nor `double`, because neither `float` nor `double` is implicitly convertible to `decimal`. You must explicitly convert the `float` or `double` operand to the `decimal` type. For more information about conversions between numeric types, see Built-in numeric conversions.

Remainder operator %

The remainder operator `%` computes the remainder after dividing its left-hand operand by its right-hand operand.

Integer remainder

For the operands of integer types, the result of `a % b` is the value produced by `a - (a / b) * b`. The sign of the non-zero remainder is the same as the sign of the left-hand operand, as the following example shows:

C#

```
Console.WriteLine(5 % 4);   // output: 1
Console.WriteLine(5 % -4);  // output: 1
Console.WriteLine(-5 % 4);  // output: -1
Console.WriteLine(-5 % -4); // output: -1
```

Use the Math.DivRem method to compute both integer division and remainder results.

Floating-point remainder

For the `float` and `double` operands, the result of `x % y` for the finite `x` and `y` is the value `z` such that

- The sign of `z`, if non-zero, is the same as the sign of `x`.
- The absolute value of `z` is the value produced by `|x| - n * |y|` where `n` is the largest possible integer that is less than or equal to `|x| / |y|` and `|x|` and `|y|` are the absolute values of `x` and `y`, respectively.

> **Note**
>
> This method of computing the remainder is analogous to the method used for integer operands, but different from the IEEE 754 specification. If you need the remainder operation that complies with the IEEE 754 specification, use the Math.IEEERemainder method.

For information about the behavior of the `%` operator with non-finite operands, see the Remainder operator section of the C# language specification.

For the `decimal` operands, the remainder operator `%` is equivalent to the remainder operator of the System.Decimal type.

The following example demonstrates the behavior of the remainder operator with floating-point operands:

C#

```
Console.WriteLine(-5.2f % 2.0f); // output: -1.2
Console.WriteLine(5.9 % 3.1);    // output: 2.8
Console.WriteLine(5.9m % 3.1m);  // output: 2.8
```

Addition operator +

The addition operator `+` computes the sum of its operands:

```
C#

Console.WriteLine(5 + 4);       // output: 9
Console.WriteLine(5 + 4.3);     // output: 9.3
Console.WriteLine(5.1m + 4.2m); // output: 9.3
```

You can also use the `+` operator for string concatenation and delegate combination. For more information, see the + and += operators article.

Subtraction operator -

The subtraction operator `-` subtracts its right-hand operand from its left-hand operand:

```
C#

Console.WriteLine(47 - 3);      // output: 44
Console.WriteLine(5 - 4.3);     // output: 0.7
Console.WriteLine(7.5m - 2.3m); // output: 5.2
```

You can also use the `-` operator for delegate removal. For more information, see the - and -= operators article.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
C#

x op= y
```

Is equivalent to

```
C#

x = x op y
```

Except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with arithmetic operators:

C#

```
int a = 5;
a += 9;
Console.WriteLine(a);  // output: 14

a -= 4;
Console.WriteLine(a);  // output: 10

a *= 2;
Console.WriteLine(a);  // output: 20

a /= 4;
Console.WriteLine(a);  // output: 5

a %= 3;
Console.WriteLine(a);  // output: 2
```

Because of numeric promotions, the result of the `op` operation might be not implicitly convertible to the type `T` of `x`. In such a case, if `op` is a predefined operator and the result of the operation is explicitly convertible to the type `T` of `x`, a compound assignment expression of the form `x op= y` is equivalent to `x = (T)(x op y)`, except that `x` is only evaluated once. The following example demonstrates that behavior:

C#

```
byte a = 200;
byte b = 100;

var c = a + b;
Console.WriteLine(c.GetType());  // output: System.Int32
Console.WriteLine(c);  // output: 300

a += b;
Console.WriteLine(a);  // output: 44
```

In the preceding example, value `44` is the result of converting value `300` to the `byte` type.

> **Note**
>
> In the checked overflow-checking context, the preceding example throws an OverflowException. For more information, see the Integer arithmetic overflow section.

You also use the `+=` and `-=` operators to subscribe to and unsubscribe from an event, respectively. For more information, see How to subscribe to and unsubscribe from events.

Operator precedence and associativity

The following list orders arithmetic operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators
- Prefix increment `++x` and decrement `--x` and unary `+` and `-` operators
- Multiplicative `*`, `/`, and `%` operators
- Additive `+` and `-` operators

Binary arithmetic operators are left-associative. That is, operators with the same precedence level are evaluated from left to right.

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence and associativity.

C#

```
Console.WriteLine(2 + 2 * 2);   // output: 6
Console.WriteLine((2 + 2) * 2); // output: 8

Console.WriteLine(9 / 5 / 2);   // output: 0
Console.WriteLine(9 / (5 / 2)); // output: 4
```

For the complete list of C# operators ordered by precedence level, see the Operator precedence section of the C# operators article.

Arithmetic overflow and division by zero

When the result of an arithmetic operation is outside the range of possible finite values of the involved numeric type, the behavior of an arithmetic operator depends on the type of its operands.

Integer arithmetic overflow

Integer division by zero always throws a DivideByZeroException.

If integer arithmetic overflow occurs, the overflow-checking context, which can be checked or unchecked, controls the resulting behavior:

- In a checked context, if overflow happens in a constant expression, a compile-time error occurs. Otherwise, when the operation is performed at run time, an OverflowException is thrown.

- In an unchecked context, the result is truncated by discarding any high-order bits that don't fit in the destination type.

Along with the checked and unchecked statements, you can use the `checked` and `unchecked` operators to control the overflow-checking context, in which an expression is evaluated:

C#

```
int a = int.MaxValue;
int b = 3;

Console.WriteLine(unchecked(a + b));  // output: -2147483646
try
{
    int d = checked(a + b);
}
catch(OverflowException)
{
    Console.WriteLine($"Overflow occurred when adding {a} to {b}.");
}
```

By default, arithmetic operations occur in an *unchecked* context.

Floating-point arithmetic overflow

Arithmetic operations with the `float` and `double` types never throw an exception. The result of arithmetic operations with those types can be one of special values that represent infinity and not-a-number:

C#

```
double a = 1.0 / 0.0;
Console.WriteLine(a);                    // output: Infinity
Console.WriteLine(double.IsInfinity(a)); // output: True

Console.WriteLine(double.MaxValue + double.MaxValue); // output: Infinity

double b = 0.0 / 0.0;
Console.WriteLine(b);                // output: NaN
Console.WriteLine(double.IsNaN(b));  // output: True
```

For the operands of the `decimal` type, arithmetic overflow always throws an OverflowException. Division by zero always throws a DivideByZeroException.

Round-off errors

Because of general limitations of the floating-point representation of real numbers and floating-point arithmetic, round-off errors might occur in calculations with floating-point types. That is, the produced result of an expression might differ from the expected mathematical result. The following example demonstrates several such cases:

```
C#
```

```
Console.WriteLine(.41f % .2f); // output: 0.00999999

double a = 0.1;
double b = 3 * a;
Console.WriteLine(b == 0.3);   // output: False
Console.WriteLine(b - 0.3);    // output: 5.55111512312578E-17

decimal c = 1 / 3.0m;
decimal d = 3 * c;
Console.WriteLine(d == 1.0m);  // output: False
Console.WriteLine(d);          // output: 0.99999999999999999999999999
```

For more information, see remarks at the System.Double, System.Single, or System.Decimal reference pages.

Operator overloadability

A user-defined type can overload the unary (`++`, `--`, `+`, and `-`) and binary (`*`, `/`, `%`, `+`, and `-`) arithmetic operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. Beginning with C# 14, a user-defined type can explicitly overload the compound assignment operators (`op=`) to provide a more efficient implementation. Typically, a type overloads these operators because the value can be updated in place, rather than allocating a new instance to store the result of the operation. If a type doesn't provide an explicit overload, the compiler generates the implicit overload.

User-defined checked operators

Beginning with C# 11, when you overload an arithmetic operator, you can use the `checked` keyword to define the *checked* version of that operator. The following example shows how to do that:

```
C#
```

```
public record struct Point(int X, int Y)
{
    public static Point operator checked +(Point left, Point right)
    {
        checked
```

```
        {
            return new Point(left.X + right.X, left.Y + right.Y);
        }
    }

    public static Point operator +(Point left, Point right)
    {
        return new Point(left.X + right.X, left.Y + right.Y);
    }
}
```

When you define a checked operator, you must also define the corresponding operator without the `checked` modifier. The checked operator is called in a checked context; the operator without the `checked` modifier is called in an unchecked context. If you only provide the operator without the `checked` modifier, it's called in both a `checked` and `unchecked` context.

When you define both versions of an operator, it's expected that their behavior differs only when the result of an operation is too large to represent in the result type as follows:

- A checked operator throws an OverflowException.
- An operator without the `checked` modifier returns an instance representing a *truncated* result.

For information about the difference in behavior of the built-in arithmetic operators, see the Arithmetic overflow and division by zero section.

You can use the `checked` modifier only when you overload any of the following operators:

- Unary `++`, `--`, and `-` operators
- Binary `*`, `/`, `+`, and `-` operators
- Compound assignment `*=`, `/=`, `+=`, and `-=` operators (C# 14 and later)
- Explicit conversion operators

> **Note**
>
> The overflow-checking context within the body of a checked operator isn't affected by the presence of the `checked` modifier. The default context is defined by the value of the CheckForOverflowUnderflow compiler option. Use the checked and unchecked statements to explicitly specify the overflow-checking context, as the example at the beginning of this section demonstrates.

C# language specification

For more information, see the following sections of the C# language specification:

- Postfix increment and decrement operators
- Prefix increment and decrement operators
- Unary plus operator
- Unary minus operator
- Multiplication operator
- Division operator
- Remainder operator
- Addition operator
- Subtraction operator
- Compound assignment
- The checked and unchecked operators
- Numeric promotions
- User defined compound assignment

See also

- C# operators and expressions
- System.Math
- System.MathF
- Numerics in .NET

Boolean logical operators - AND, OR, NOT, XOR

06/13/2025

The logical Boolean operators perform logical operations with bool operands. The operators include the unary logical negation (`!`), binary logical AND (`&`), OR (`|`), and exclusive OR (`^`), and the binary conditional logical AND (`&&`) and OR (`||`).

- Unary ! (logical negation) operator.
- Binary & (logical AND), | (logical OR), and ^ (logical exclusive OR) operators. Those operators always evaluate both operands.
- Binary && (conditional logical AND) and || (conditional logical OR) operators. Those operators evaluate the right-hand operand only if it's necessary.

For operands of the integral numeric types, the `&`, `|`, and `^` operators perform bitwise logical operations. For more information, see Bitwise and shift operators.

Logical negation operator !

The unary prefix `!` operator computes logical negation of its operand. That is, it produces `true`, if the operand evaluates to `false`, and `false`, if the operand evaluates to `true`:

C#

```
bool passed = false;
Console.WriteLine(!passed);  // output: True
Console.WriteLine(!true);    // output: False
```

The unary postfix `!` operator is the null-forgiving operator.

Logical AND operator &

The `&` operator computes the logical AND of its operands. The result of `x & y` is `true` if both `x` and `y` evaluate to `true`. Otherwise, the result is `false`.

The `&` operator always evaluates both operands. When the left-hand operand evaluates to `false`, the operation result is `false` regardless of the value of the right-hand operand. However, even then, the right-hand operand is evaluated.

In the following example, the right-hand operand of the `&` operator is a method call, which is performed regardless of the value of the left-hand operand:

```
C#
```

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false & SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// False

bool b = true & SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The conditional logical AND operator `&&` also computes the logical AND of its operands, but doesn't evaluate the right-hand operand if the left-hand operand evaluates to `false`.

For operands of the integral numeric types, the `&` operator computes the bitwise logical AND of its operands. The unary `&` operator is the address-of operator.

Logical exclusive OR operator ^

The `^` operator computes the logical exclusive OR, also known as the logical XOR, of its operands. The result of `x ^ y` is `true` if `x` evaluates to `true` and `y` evaluates to `false`, or `x` evaluates to `false` and `y` evaluates to `true`. Otherwise, the result is `false`. That is, for the `bool` operands, the `^` operator computes the same result as the inequality operator `!=`.

```
C#
```

```
Console.WriteLine(true ^ true);    // output: False
Console.WriteLine(true ^ false);   // output: True
Console.WriteLine(false ^ true);   // output: True
Console.WriteLine(false ^ false);  // output: False
```

For operands of the integral numeric types, the `^` operator computes the bitwise logical exclusive OR of its operands.

Logical OR operator |

The `|` operator computes the logical OR of its operands. The result of `x | y` is `true` if either `x` or `y` evaluates to `true`. Otherwise, the result is `false`.

The `|` operator always evaluates both operands. When the left-hand operand evaluates to `true`, the operation result is `true` regardless of the value of the right-hand operand. However, even then, the right-hand operand is evaluated.

In the following example, the right-hand operand of the `|` operator is a method call, which is performed regardless of the value of the left-hand operand:

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true | SecondOperand();
Console.WriteLine(a);
// Output:
// Second operand is evaluated.
// True

bool b = false | SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The conditional logical OR operator `||` also computes the logical OR of its operands, but doesn't evaluate the right-hand operand if the left-hand operand evaluates to `true`.

For operands of the integral numeric types, the `|` operator computes the bitwise logical OR of its operands.

Conditional logical AND operator &&

The conditional logical AND operator `&&`, also known as the "short-circuiting" logical AND operator, computes the logical AND of its operands. The result of `x && y` is `true` if both `x` and `y` evaluate to `true`. Otherwise, the result is `false`. If `x` evaluates to `false`, `y` isn't evaluated.

In the following example, the right-hand operand of the `&&` operator is a method call, which isn't performed if the left-hand operand evaluates to `false`:

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = false && SecondOperand();
Console.WriteLine(a);
// Output:
// False

bool b = true && SecondOperand();
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The logical AND operator `&` also computes the logical AND of its operands, but always evaluates both operands.

Conditional logical OR operator ||

The conditional logical OR operator `||`, also known as the "short-circuiting" logical OR operator, computes the logical OR of its operands. The result of `x || y` is `true` if either `x` or `y` evaluates to `true`. Otherwise, the result is `false`. If `x` evaluates to `true`, `y` isn't evaluated.

In the following example, the right-hand operand of the `||` operator is a method call, which isn't performed if the left-hand operand evaluates to `true`:

C#

```
bool SecondOperand()
{
    Console.WriteLine("Second operand is evaluated.");
    return true;
}

bool a = true || SecondOperand();
Console.WriteLine(a);
// Output:
// True

bool b = false || SecondOperand();
```

```
Console.WriteLine(b);
// Output:
// Second operand is evaluated.
// True
```

The logical OR operator `|` also computes the logical OR of its operands, but always evaluates both operands.

Nullable Boolean logical operators

For `bool?` operands, the & (logical AND) and | (logical OR) operators support the three-valued logic as follows:

- The `&` operator produces `true` only if both its operands evaluate to `true`. If either `x` or `y` evaluates to `false`, `x & y` produces `false` (even if another operand evaluates to `null`). Otherwise, the result of `x & y` is `null`.
- The `|` operator produces `false` only if both its operands evaluate to `false`. If either `x` or `y` evaluates to `true`, `x | y` produces `true` (even if another operand evaluates to `null`). Otherwise, the result of `x | y` is `null`.

The following table presents that semantics:

Expand table

x	y	x & y	x | y
true	true	true	true
true	false	false	true
true	null	null	true
false	true	false	true
false	false	false	false
false	null	false	null
null	true	null	true
null	false	false	null
null	null	null	null

The behavior of those operators differs from the typical operator behavior with nullable value types. Typically, an operator that is defined for operands of a value type can be also used with operands of the corresponding nullable value type. Such an operator produces `null` if any of its operands evaluates to `null`. However, the `&` and `|` operators can produce non-null even if one of the operands evaluates to `null`. For more information about the operator behavior with nullable value types, see the Lifted operators section of the Nullable value types article.

You can also use the `!` and `^` operators with `bool?` operands, as the following example shows:

```
C#

bool? test = null;
Display(!test);         // output: null
Display(test ^ false);  // output: null
Display(test ^ null);   // output: null
Display(true ^ null);   // output: null

void Display(bool? b) => Console.WriteLine(b is null ? "null" :
b.Value.ToString());
```

The conditional logical operators `&&` and `||` don't support `bool?` operands.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```
C#

x op= y
```

Is equivalent to

```
C#

x = x op y
```

Except that `x` is only evaluated once.

The `&`, `|`, and `^` operators support compound assignment, as the following example shows:

```
C#

bool test = true;
test &= false;
Console.WriteLine(test);  // output: False
```

```
test |= true;
Console.WriteLine(test);  // output: True

test ^= false;
Console.WriteLine(test);  // output: True
```

> **Note**
>
> The conditional logical operators `&&` and `||` don't support compound assignment.

Operator precedence

The following list orders logical operators starting from the highest precedence to the lowest:

- Logical negation operator `!`
- Logical AND operator `&`
- Logical exclusive OR operator `^`
- Logical OR operator `|`
- Conditional logical AND operator `&&`
- Conditional logical OR operator `||`

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence:

C#

```
Console.WriteLine(true | true & false);   // output: True
Console.WriteLine((true | true) & false); // output: False

bool Operand(string name, bool value)
{
    Console.WriteLine($"Operand {name} is evaluated.");
    return value;
}

var byDefaultPrecedence = Operand("A", true) || Operand("B", true) && Operand("C",
false);
Console.WriteLine(byDefaultPrecedence);
// Output:
// Operand A is evaluated.
// True

var changedOrder = (Operand("A", true) || Operand("B", true)) && Operand("C",
false);
Console.WriteLine(changedOrder);
// Output:
```

```
// Operand A is evaluated.
// Operand C is evaluated.
// False
```

For the complete list of C# operators ordered by precedence level, see the Operator precedence section of the C# operators article.

Operator overloadability

A user-defined type can overload the `!`, `&`, `|`, and `^` operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. Beginning with C# 14, a user-defined type can explicitly overload the compound assignment operators to provide a more efficient implementation. Typically, a type overloads these operators because the value can be updated in place, rather than allocating a new instance to hold the result of the binary operation. If a type doesn't provide an explicit overload, the compiler generates the implicit overload.

A user-defined type can't overload the conditional logical operators `&&` and `||`. However, if a user-defined type overloads the true and false operators and the `&` or `|` operator in a certain way, the `&&` or `||` operation, respectively, can be evaluated for the operands of that type. For more information, see the User-defined conditional logical operators section of the C# language specification.

C# language specification

For more information, see the following sections of the C# language specification:

- Logical negation operator
- Logical operators
- Conditional logical operators
- Compound assignment
- User defined compound assignment

See also

- C# operators and expressions
- Bitwise and shift operators

Bitwise and shift operators (C# reference)

06/11/2025

The bitwise and shift operators include unary bitwise complement, binary left and right shift, unsigned right shift, and the binary logical AND, OR, and exclusive OR operators. These operands take operands of the integral numeric types or the char type.

- Unary ~ (bitwise complement) operator
- Binary << (left shift), >> (right shift), and >>> (unsigned right shift) operators
- Binary & (logical AND), | (logical OR), and ^ (logical exclusive OR) operators

Those operators are defined for the `int`, `uint`, `long`, `ulong`, `nint`, and `nuint` types. When both operands are of other integral types (`sbyte`, `byte`, `short`, `ushort`, or `char`), their values are converted to the `int` type, which is also the result type of an operation. When operands are of different integral types, their values are converted to the closest containing integral type. For more information, see the Numeric promotions section of the C# language specification. The compound operators (such as `>>=`) don't convert their arguments to `int` or have the result type as `int`.

The `&`, `|`, and `^` operators are also defined for operands of the `bool` type. For more information, see Boolean logical operators.

Bitwise and shift operations never cause overflow and produce the same results in checked and unchecked contexts.

Bitwise complement operator ~

The `~` operator produces a bitwise complement of its operand by reversing each bit:

C#

```
uint a = 0b_0000_1111_0000_1111_0000_1111_0000_1100;
uint b = ~a;
Console.WriteLine(Convert.ToString(b, toBase: 2));
// Output:
// 11110000111100001111000011110011
```

You can also use the `~` symbol to declare finalizers. For more information, see Finalizers.

Left-shift operator <<

The `<<` operator shifts its left-hand operand left by the number of bits defined by its right-hand operand. For information about how the right-hand operand defines the shift count, see the Shift count of the shift operators section.

The left-shift operation discards the high-order bits that are outside the range of the result type and sets the low-order empty bit positions to zero, as the following example shows:

```
C#
```

```
uint x = 0b_1100_1001_0000_0000_0000_0000_0001_0001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2)}");

uint y = x << 4;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2)}");
// Output:
// Before: 11001001000000000000000000010001
// After:  10010000000000000000000100010000
```

Because the shift operators are defined only for the `int`, `uint`, `long`, and `ulong` types, the result of an operation always contains at least 32 bits. If the left-hand operand is of another integral type (`sbyte`, `byte`, `short`, `ushort`, or `char`), its value is converted to the `int` type, as the following example shows:

```
C#
```

```
byte a = 0b_1111_0001;

var b = a << 8;
Console.WriteLine(b.GetType());
Console.WriteLine($"Shifted byte: {Convert.ToString(b, toBase: 2)}");
// Output:
// System.Int32
// Shifted byte: 1111000100000000
```

Right-shift operator >>

The `>>` operator shifts its left-hand operand right by the number of bits defined by its right-hand operand. For information about how the right-hand operand defines the shift count, see the Shift count of the shift operators section.

The right-shift operation discards the low-order bits, as the following example shows:

```
C#
```

```
uint x = 0b_1001;
Console.WriteLine($"Before: {Convert.ToString(x, toBase: 2), 4}");
```

```
uint y = x >> 2;
Console.WriteLine($"After:  {Convert.ToString(y, toBase: 2).PadLeft(4, '0'), 4}");
// Output:
// Before: 1001
// After:  0010
```

The high-order empty bit positions are set based on the type of the left-hand operand as follows:

- If the left-hand operand is of type `int` or `long`, the right-shift operator performs an *arithmetic* shift: the value of the most significant bit (the sign bit) of the left-hand operand is propagated to the high-order empty bit positions. That is, the high-order empty bit positions are set to zero if the left-hand operand is non-negative and set to one if it's negative.

  C#

  ```
  int a = int.MinValue;
  Console.WriteLine($"Before: {Convert.ToString(a, toBase: 2)}");

  int b = a >> 3;
  Console.WriteLine($"After:  {Convert.ToString(b, toBase: 2)}");
  // Output:
  // Before: 10000000000000000000000000000000
  // After:  11110000000000000000000000000000
  ```

- If the left-hand operand is of type `uint` or `ulong`, the right-shift operator performs a *logical* shift: the high-order empty bit positions are always set to zero.

  C#

  ```
  uint c = 0b_1000_0000_0000_0000_0000_0000_0000_0000;
  Console.WriteLine($"Before: {Convert.ToString(c, toBase: 2), 32}");

  uint d = c >> 3;
  Console.WriteLine($"After:  {Convert.ToString(d, toBase: 2).PadLeft(32, '0'),
  32}");
  // Output:
  // Before: 10000000000000000000000000000000
  // After:  00010000000000000000000000000000
  ```

> **Note**
>
> Use the unsigned right-shift operator to perform a *logical* shift on operands of signed integer types. The logical shift is preferred to casting a left-hand operand to an unsigned

type and then casting the result of a shift operation back to a signed type.

Unsigned right-shift operator >>>

Available in C# 11 and later, the `>>>` operator shifts its left-hand operand right by the number of bits defined by its right-hand operand. For information about how the right-hand operand defines the shift count, see the Shift count of the shift operators section.

The `>>>` operator always performs a *logical* shift. That is, the high-order empty bit positions are always set to zero, regardless of the type of the left-hand operand. The `>>` operator performs an *arithmetic* shift (that is, the value of the most significant bit is propagated to the high-order empty bit positions) if the left-hand operand is of a signed type. The following example demonstrates the difference between `>>` and `>>>` operators for a negative left-hand operand:

C#

```
int x = -8;
Console.WriteLine($"Before:    {x,11}, hex: {x,8:x}, binary: {Convert.ToString(x,
toBase: 2), 32}");

int y = x >> 2;
Console.WriteLine($"After  >>: {y,11}, hex: {y,8:x}, binary: {Convert.ToString(y,
toBase: 2), 32}");

int z = x >>> 2;
Console.WriteLine($"After >>>: {z,11}, hex: {z,8:x}, binary: {Convert.ToString(z,
toBase: 2).PadLeft(32, '0'), 32}");
// Output:
// Before:             -8, hex: fffffff8, binary: 11111111111111111111111111111000
// After  >>:          -2, hex: fffffffe, binary: 11111111111111111111111111111110
// After >>>:  1073741822, hex: 3ffffffe, binary: 00111111111111111111111111111110
```

Logical AND operator &

The `&` operator computes the bitwise logical AND of its integral operands:

C#

```
uint a = 0b_1111_1000;
uint b = 0b_1001_1101;
uint c = a & b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10011000
```

For `bool` operands, the `&` operator computes the logical AND of its operands. The unary `&` operator is the address-of operator.

Logical exclusive OR operator ^

The `^` operator computes the bitwise logical exclusive OR, also known as the bitwise logical XOR, of its integral operands:

C#

```
uint a = 0b_1111_1000;
uint b = 0b_0001_1100;
uint c = a ^ b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 11100100
```

For `bool` operands, the `^` operator computes the logical exclusive OR of its operands.

Logical OR operator |

The `|` operator computes the bitwise logical OR of its integral operands:

C#

```
uint a = 0b_1010_0000;
uint b = 0b_1001_0001;
uint c = a | b;
Console.WriteLine(Convert.ToString(c, toBase: 2));
// Output:
// 10110001
```

For `bool` operands, the `|` operator computes the logical OR of its operands.

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

C#

```
x op= y
```

Is equivalent to

C#

```
x = x op y
```

Except that `x` is only evaluated once.

The following example demonstrates the usage of compound assignment with bitwise and shift operators:

C#

```
uint INITIAL_VALUE = 0b_1111_1000;

uint a = INITIAL_VALUE;
a &= 0b_1001_1101;
Display(a);  // output: 10011000

a = INITIAL_VALUE;
a |= 0b_0011_0001;
Display(a);  // output: 11111001

a = INITIAL_VALUE;
a ^= 0b_1000_0000;
Display(a);  // output: 01111000

a = INITIAL_VALUE;
a <<= 2;
Display(a);  // output: 1111100000

a = INITIAL_VALUE;
a >>= 4;
Display(a);  // output: 00001111

a = INITIAL_VALUE;
a >>>= 4;
Display(a);  // output: 00001111

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase:
2).PadLeft(8, '0'), 8}");
```

Because of numeric promotions, the result of the `op` operation might be not implicitly convertible to the type `T` of `x`. In such a case, if `op` is a predefined operator and the result of the operation is explicitly convertible to the type `T` of `x`, a compound assignment expression of the form `x op= y` is equivalent to `x = (T)(x op y)`, except that `x` is only evaluated once. The following example demonstrates that behavior:

C#

```
byte x = 0b_1111_0001;

int b = x << 8;
Console.WriteLine($"{Convert.ToString(b, toBase: 2)}");  // output:
1111000100000000

x <<= 8;
Console.WriteLine(x);  // output: 0
```

Operator precedence

The following list orders bitwise and shift operators starting from the highest precedence to the lowest:

- Bitwise complement operator `~`
- Shift operators `<<`, `>>`, and `>>>`
- Logical AND operator `&`
- Logical exclusive OR operator `^`
- Logical OR operator `|`

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence:

C#

```
uint a = 0b_1101;
uint b = 0b_1001;
uint c = 0b_1010;

uint d1 = a | b & c;
Display(d1);  // output: 1101

uint d2 = (a | b) & c;
Display(d2);  // output: 1000

void Display(uint x) => Console.WriteLine($"{Convert.ToString(x, toBase: 2), 4}");
```

For the complete list of C# operators ordered by precedence level, see the Operator precedence section of the C# operators article.

Shift count of the shift operators

For the `x << count`, `x >> count`, and `x >>> count` expressions, the actual shift count depends on the type of `x` as follows:

- If the type of `x` is `int` or `uint`, the low-order *five* bits of the right-hand operand define the shift count. That is, the shift count is computed from `count & 0x1F` (or `count & 0b_1_1111`).
- If the type of `x` is `long` or `ulong`, the low-order *six* bits of the right-hand operand define the shift count. That is, the shift count is computed from `count & 0x3F` (or `count & 0b_11_1111`).

The following example demonstrates that behavior:

C#

```
int count1 = 0b_0000_0001;
int count2 = 0b_1110_0001;

int a = 0b_0001;
Console.WriteLine($"{a} << {count1} is {a << count1}; {a} << {count2} is {a <<
count2}");
// Output:
// 1 << 1 is 2; 1 << 225 is 2

int b = 0b_0100;
Console.WriteLine($"{b} >> {count1} is {b >> count1}; {b} >> {count2} is {b >>
count2}");
// Output:
// 4 >> 1 is 2; 4 >> 225 is 2

int count = -31;
int c = 0b_0001;
Console.WriteLine($"{c} << {count} is {c << count}");
// Output:
// 1 << -31 is 2
```

> **Note**
>
> As the preceding example shows, the result of a shift operation can be non-zero even if the value of the right-hand operand is greater than the number of bits in the left-hand operand.

Enumeration logical operators

Every enumeration type supports the `~`, `&`, `|`, and `^` operators. For operands of the same enumeration type, a logical operation is performed on the corresponding values of the underlying integral type. For example, for any `x` and `y` of an enumeration type `T` with an

underlying type `U`, the `x & y` expression produces the same result as the `(T)((U)x & (U)y)` expression.

You typically use bitwise logical operators with an enumeration type that is defined with the Flags attribute. For more information, see the Enumeration types as bit flags section of the Enumeration types article.

Operator overloadability

A user-defined type can overload the `~`, `<<`, `>>`, `>>>`, `&`, `|`, and `^` operators. When a binary operator is overloaded, the corresponding compound assignment operator is also implicitly overloaded. Beginning with C# 14, a user-defined type can explicitly overload the compound assignment operators to provide a more efficient implementation. Typically, a type overloads these operators because the value can be updated in place, rather than allocating a new instance to store the result of the binary operation. If a type doesn't provide an explicit overload, the compiler generates the implicit overload.

If a user-defined type `T` overloads the `<<`, `>>`, or `>>>` operator, the type of the left-hand operand must be `T`. In C# 10 and earlier, the type of the right-hand operand must be `int`; beginning with C# 11, the type of the right-hand operand of an overloaded shift operator can be any.

C# language specification

For more information, see the following sections of the C# language specification:

- Bitwise complement operator
- Shift operators
- Logical operators
- Compound assignment
- Numeric promotions
- C# 11 - Relaxed shift requirements
- C# 11 - Logical right-shift operator
- User defined compound assignment

See also

- C# operators and expressions
- Boolean logical operators

Collection expressions - C# language reference

Article • 03/12/2024

You can use a *collection expression* to create common collection values. A *collection expression* is a terse syntax that, when evaluated, can be assigned to many different collection types. A collection expression contains a sequence of elements between `[` and `]` brackets. The following example declares a System.Span<T> of `string` elements and initializes them to the days of the week:

C#

```
Span<string> weekDays = ["Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat"];
foreach (var day in weekDays)
{
    Console.WriteLine(day);
}
```

A *collection expression* can be converted to many different collection types. The first example demonstrated how to initialize a variable using a collection expression. The following code shows many of the other locations where you can use a collection expression:

C#

```
// Initialize private field:
private static readonly ImmutableArray<string> _months = ["Jan", "Feb",
"Mar", "Apr", "May", "Jun", "Jul", "Aug", "Sep", "Oct", "Nov", "Dec"];

// property with expression body:
public IEnumerable<int> MaxDays =>
    [31, 28, 31, 30, 31, 30, 31, 31, 30, 31, 30, 31];

public int Sum(IEnumerable<int> values) =>
    values.Sum();

public void Example()
{
    // As a parameter:
    int sum = Sum([1, 2, 3, 4, 5]);
}
```

You can't use a collection expression where a compile-time constant is expected, such as initializing a constant, or as the default value for a method argument.

Both of the previous examples used constants as the elements of a collection expression. You can also use variables for the elements as shown in the following example:

```
C#
```

```
string hydrogen = "H";
string helium = "He";
string lithium = "Li";
string beryllium = "Be";
string boron = "B";
string carbon = "C";
string nitrogen = "N";
string oxygen = "O";
string fluorine = "F";
string neon = "Ne";
string[] elements = [hydrogen, helium, lithium, beryllium, boron, carbon,
nitrogen, oxygen, fluorine, neon];
foreach (var element in elements)
{
    Console.WriteLine(element);
}
```

Spread element

You use a *spread element* `..` to inline collection values in a collection expression. The following example creates a collection for the full alphabet by combining a collection of the vowels, a collection of the consonants, and the letter "y", which can be either:

```
C#
```

```
string[] vowels = ["a", "e", "i", "o", "u"];
string[] consonants = ["b", "c", "d", "f", "g", "h", "j", "k", "l", "m",
                       "n", "p", "q", "r", "s", "t", "v", "w", "x", "z"];
string[] alphabet = [.. vowels, .. consonants, "y"];
```

The spread element `..vowels`, when evaluated, produces five elements: `"a"`, `"e"`, `"i"`, `"o"`, and `"u"`. The spread element `..consonants` produces 20 elements, the number in the `consonants` array. The variable in a spread element must be enumerable using a foreach statement. As shown in the previous example, you can combine spread elements with individual elements in a collection expression.

Conversions

A *collection expression* can be converted to different collection types, including:

- System.Span<T> and System.ReadOnlySpan<T>.
- Arrays.
- Any type with a *create* method whose parameter type is `ReadOnlySpan<T>` where there's an implicit conversion from the collection expression type to `T`.
- Any type that supports a collection initializer, such as System.Collections.Generic.List<T>. Usually, this requirement means the type supports System.Collections.Generic.IEnumerable<T> and there's an accessible `Add` method to add items to the collection. There must be an implicit conversion from the collection expression elements' type to the collection's element type. For spread elements, there must be an implicit conversion from the spread element's type to the collection's element type.
- Any of the following interfaces:
  - System.Collections.Generic.IEnumerable<T>.
  - System.Collections.Generic.IReadOnlyCollection<T>.
  - System.Collections.Generic.IReadOnlyList<T>.
  - System.Collections.Generic.ICollection<T>.
  - System.Collections.Generic.IList<T>.

> **Important**
>
> A collection expression always creates a collection that includes all elements in the collection expression, regardless of the target type of the conversion. For example, when the target of the conversion is **System.Collections.Generic.IEnumerable<T>**, the generated code evaluates the collection expression and stores the results in an in-memory collection.
>
> This behavior is distinct from LINQ, where a sequence might not be instantiated until it is enumerated. You can't use collection expressions to generate an infinite sequence that won't be enumerated.

The compiler uses static analysis to determine the most performant way to create the collection declared with a collection expression. For example, the empty collection expression, `[]`, can be realized as Array.Empty<T>() if the target won't be modified after initialization. When the target is a System.Span<T> or System.ReadOnlySpan<T>, the storage might be stack allocated. The collection expressions feature specification specifies the rules the compiler must follow.

Many APIs are overloaded with multiple collection types as parameters. Because a collection expression can be converted to many different expression types, these APIs might require casts on the collection expression to specify the correct conversion. The following conversion rules resolve some of the ambiguities:

- A better element conversion is preferred over a better collection type conversion. In other words, the type of elements in the collection expression has more importance than the type of the collection. These rules are described in the feature spec for better conversion from collection expression.
- Conversion to Span<T>, ReadOnlySpan<T>, or another ref struct type is better than a conversion to a non-ref struct type.
- Conversion to a noninterface type is better than a conversion to an interface type.

When a collection expression is converted to a `Span` or `ReadOnlySpan`, the span object's *safe context* is taken from the *safe context* of all elements included in the span. For detailed rules, see the Collection expression specification.

Collection builder

Collection expressions work with any collection type that's *well-behaved*. A well-behaved collection has the following characteristics:

- The value of `Count` or `Length` on a countable collection produces the same value as the number of elements when enumerated.
- The types in the System.Collections.Generic namespace are presumed to be side-effect free. As such, the compiler can optimize scenarios where such types might be used as intermediary values, but otherwise not be exposed.
- A call to some applicable `.AddRange(x)` member on a collection will result in the same final value as iterating over `x` and adding all of its enumerated values individually to the collection with `.Add`.

All the collection types in the .NET runtime are well-behaved.

> **⚠ Warning**
>
> If a custom collection type isn't well-behaved, the behavior when you use that collection type with collection expressions is undefined.

Your types opt in to collection expression support by writing a `Create()` method and applying the System.Runtime.CompilerServices.CollectionBuilderAttribute on the collection type to indicate the builder method. For example, consider an application that uses fixed length buffers of 80 characters. That class might look something like the following code:

```
C#
```

```
public class LineBuffer : IEnumerable<char>
{
    private readonly char[] _buffer = new char[80];

    public LineBuffer(ReadOnlySpan<char> buffer)
    {
        int number = (_buffer.Length < buffer.Length) ? _buffer.Length :
buffer.Length;
        for (int i = 0; i < number; i++)
        {
            _buffer[i] = buffer[i];
        }
    }

    public IEnumerator<char> GetEnumerator() => _buffer.AsEnumerable<char>
().GetEnumerator();
    IEnumerator IEnumerable.GetEnumerator() => _buffer.GetEnumerator();

    // etc
}
```

You'd like to use it with collection expressions as shown in the following sample:

C#

```
LineBuffer line = ['H', 'e', 'l', 'l', 'o', ' ', 'W', 'o', 'r', 'l', 'd',
'!'];
```

The `LineBuffer` type implements `IEnumerable<char>`, so the compiler recognizes it as a collection of `char` items. The type parameter of the implemented System.Collections.Generic.IEnumerable<T> interface indicates the element type. You need to make two additions to your application to be able to assign collection expressions to a `LineBuffer` object. First, you need to create a class that contains a `Create` method:

C#

```
internal static class LineBufferBuilder
{
    internal static LineBuffer Create(ReadOnlySpan<char> values) => new
LineBuffer(values);
}
```

The `Create` method must return a `LineBuffer` object, and it must take a single parameter of the type `ReadOnlySpan<char>`. The type parameter of the `ReadOnlySpan` must match the element type of the collection. A builder method that returns a generic

collection would have the generic `ReadOnlySpan<T>` as its parameter. The method must be accessible and `static`.

Finally, you must add the CollectionBuilderAttribute to the `LineBuffer` class declaration:

```
C#

[CollectionBuilder(typeof(LineBufferBuilder), "Create")]
```

The first parameter provides the name of the *Builder* class. The second attribute provides the name of the builder method.

Equality operators - test if two objects are equal or not

Article • 02/19/2025

The == (equality) and != (inequality) operators check if their operands are equal or not. Value types are equal when their contents are equal. Reference types are equal when the two variables refer to the same storage.

You can use the is pattern matching operator as an alternative to an `==` test when you test against a constant value. The `is` operator uses the default equality semantics for all value and reference types.

Equality operator ==

The equality operator `==` returns `true` if its operands are equal, `false` otherwise.

Value types equality

Operands of the built-in value types are equal if their values are equal:

C#

```
int a = 1 + 2 + 3;
int b = 6;
Console.WriteLine(a == b);  // output: True

char c1 = 'a';
char c2 = 'A';
Console.WriteLine(c1 == c2);  // output: False
Console.WriteLine(c1 == char.ToLower(c2));  // output: True
```

> **Note**
>
> For the `==`, <, >, <=, and >= operators, if any of the operands isn't a number (Double.NaN or Single.NaN), the result of operation is `false`. That means that the `NaN` value isn't greater than, less than, or equal to any other `double` (or `float`) value, including `NaN`. For more information and examples, see the Double.NaN or Single.NaN reference article.

Two operands of the same enum type are equal if the corresponding values of the underlying integral type are equal.

User-defined struct types don't support the `==` operator by default. To support the `==` operator, a user-defined struct must overload it.

C# tuples have built-in support for the `==` and `!=` operators. For more information, see the Tuple equality section of the Tuple types article.

Reference types equality

By default, reference-type operands, excluding records, are equal if they refer to the same object:

C#

```
public class ReferenceTypesEquality
{
    public class MyClass
    {
        private int id;

        public MyClass(int id) => this.id = id;
    }

    public static void Main()
    {
        var a = new MyClass(1);
        var b = new MyClass(1);
        var c = a;
        Console.WriteLine(a == b);  // output: False
        Console.WriteLine(a == c);  // output: True
    }
}
```

As the example shows, user-defined reference types support the `==` operator by default. However, a reference type can overload the `==` operator. If a reference type overloads the `==` operator, use the Object.ReferenceEquals method to check if two references of that type refer to the same object.

Record types equality

Record types support the `==` and `!=` operators that by default provide value equality semantics. That is, two record operands are equal when both of them are `null` or corresponding values of all fields and automatically implemented properties are equal.

C#

```
public class RecordTypesEquality
{
    public record Point(int X, int Y, string Name);
    public record TaggedNumber(int Number, List<string> Tags);

    public static void Main()
    {
        var p1 = new Point(2, 3, "A");
        var p2 = new Point(1, 3, "B");
        var p3 = new Point(2, 3, "A");

        Console.WriteLine(p1 == p2);  // output: False
        Console.WriteLine(p1 == p3);  // output: True

        var n1 = new TaggedNumber(2, new List<string>() { "A" });
        var n2 = new TaggedNumber(2, new List<string>() { "A" });
        Console.WriteLine(n1 == n2);  // output: False
    }
}
```

As the preceding example shows, for reference-type members their reference values are compared, not the referenced instances.

String equality

Two string operands are equal when both of them are `null` or both string instances are of the same length and have identical characters in each character position:

C#

```
string s1 = "hello!";
string s2 = "HeLLo!";
Console.WriteLine(s1 == s2.ToLower());  // output: True

string s3 = "Hello!";
Console.WriteLine(s1 == s3);  // output: False
```

String equality comparisons are case-sensitive ordinal comparisons. For more information about string comparison, see How to compare strings in C#.

Delegate equality

Two delegate operands of the same run-time type are equal when both of them are `null` or their invocation lists are of the same length and have equal entries in each position:

C#

```
Action a = () => Console.WriteLine("a");

Action b = a + a;
Action c = a + a;
Console.WriteLine(object.ReferenceEquals(b, c));  // output: False
Console.WriteLine(b == c);  // output: True
```

> **Important**
>
> Equal entries in an invocation list include all fixed parameters in the invocation, including the receiver. The receiver is the instance of an object represented by `this` when the entry is invoked.

C#

```
var o1 = new object();
var o2 = new object();
var d1 = o1.ToString;
var d2 = o2.ToString;
Console.WriteLine(object.ReferenceEquals(d1, d2));  // output: False
Console.WriteLine(d1 == d2);  // output: False (different receivers)
```

For more information, see the Delegate equality operators section of the C# language specification.

Delegates that are produced from evaluation of semantically identical lambda expressions aren't equal, as the following example shows:

C#

```
Action a = () => Console.WriteLine("a");
Action b = () => Console.WriteLine("a");

Console.WriteLine(a == b);  // output: False
Console.WriteLine(a + b == a + b);  // output: True
Console.WriteLine(b + a == a + b);  // output: False
```

Inequality operator `!=`

The inequality operator `!=` returns `true` if its operands aren't equal, `false` otherwise. For the operands of the built-in types, the expression `x != y` produces the same result

as the expression `!(x == y)`. For more information about type equality, see the Equality operator section.

The following example demonstrates the usage of the `!=` operator:

```
C#
```

```
int a = 1 + 1 + 2 + 3;
int b = 6;
Console.WriteLine(a != b);  // output: True

string s1 = "Hello";
string s2 = "Hello";
Console.WriteLine(s1 != s2);  // output: False

object o1 = 1;
object o2 = 1;
Console.WriteLine(o1 != o2);  // output: True
```

Operator overloadability

A user-defined type can overload the `==` and `!=` operators. If a type overloads one of the two operators, it must also overload the other one.

A record type can't explicitly overload the `==` and `!=` operators. If you need to change the behavior of the `==` and `!=` operators for record type `T`, implement the IEquatable<T>.Equals method with the following signature:

```
C#
```

```
public virtual bool Equals(T? other);
```

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

For more information about equality of record types, see the Equality members section of the records feature proposal note.

See also

- C# operators and expressions

- System.IEquatable<T>
- Object.Equals
- Object.ReferenceEquals
- Equality comparisons
- Comparison operators

Comparison operators (C# reference)

Article • 04/08/2023

The < (less than), > (greater than), <= (less than or equal), and >= (greater than or equal) comparison, also known as relational, operators compare their operands. Those operators are supported by all integral and floating-point numeric types.

> **Note**
>
> For the `==`, `<`, `>`, `<=`, and `>=` operators, if any of the operands is not a number (Double.NaN or Single.NaN), the result of operation is `false`. That means that the `NaN` value is neither greater than, less than, nor equal to any other `double` (or `float`) value, including `NaN`. For more information and examples, see the Double.NaN or Single.NaN reference article.

The char type also supports comparison operators. In the case of `char` operands, the corresponding character codes are compared.

Enumeration types also support comparison operators. For operands of the same enum type, the corresponding values of the underlying integral type are compared.

The == and != operators check if their operands are equal or not.

Less than operator <

The `<` operator returns `true` if its left-hand operand is less than its right-hand operand, `false` otherwise:

C#

```
Console.WriteLine(7.0 < 5.1);   // output: False
Console.WriteLine(5.1 < 5.1);   // output: False
Console.WriteLine(0.0 < 5.1);   // output: True

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Greater than operator >

The `>` operator returns `true` if its left-hand operand is greater than its right-hand operand, `false` otherwise:

```
C#

Console.WriteLine(7.0 > 5.1);   // output: True
Console.WriteLine(5.1 > 5.1);   // output: False
Console.WriteLine(0.0 > 5.1);   // output: False

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Less than or equal operator <=

The `<=` operator returns `true` if its left-hand operand is less than or equal to its right-hand operand, `false` otherwise:

```
C#

Console.WriteLine(7.0 <= 5.1);   // output: False
Console.WriteLine(5.1 <= 5.1);   // output: True
Console.WriteLine(0.0 <= 5.1);   // output: True

Console.WriteLine(double.NaN > 5.1);   // output: False
Console.WriteLine(double.NaN <= 5.1);  // output: False
```

Greater than or equal operator >=

The `>=` operator returns `true` if its left-hand operand is greater than or equal to its right-hand operand, `false` otherwise:

```
C#

Console.WriteLine(7.0 >= 5.1);   // output: True
Console.WriteLine(5.1 >= 5.1);   // output: True
Console.WriteLine(0.0 >= 5.1);   // output: False

Console.WriteLine(double.NaN < 5.1);   // output: False
Console.WriteLine(double.NaN >= 5.1);  // output: False
```

Operator overloadability

A user-defined type can overload the `<`, `>`, `<=`, and `>=` operators.

If a type overloads one of the `<` or `>` operators, it must overload both `<` and `>`. If a type overloads one of the `<=` or `>=` operators, it must overload both `<=` and `>=`.

C# language specification

For more information, see the Relational and type-testing operators section of the C# language specification.

See also

- C# operators and expressions
- System.IComparable<T>
- Equality operators

Member access operators and expressions - the dot, indexer, and invocation operators.

05/29/2025

You use several operators and expressions to access a type member. Member access operators include member access (`.`), array element, or indexer access (`[]`), index-from-end (`^`), range (`..`), null-conditional operators (`?.` and `?[]`), and method invocation (`()`). These operators include the *null-conditional* member access (`?.`), and indexer access (`?[]`) operators.

- . (member access): to access a member of a namespace or a type
- [] (array element or indexer access): to access an array element or a type indexer
- ?. and ?[] (null-conditional operators): to perform a member or element access operation only if an operand is non-null
- () (invocation): to call an accessed method or invoke a delegate
- ^ (index from end): to indicate that the element position is from the end of a sequence
- .. (range): to specify a range of indices that you can use to obtain a range of sequence elements

Member access expression `.`

You use the `.` token to access a member of a namespace or a type, as the following examples demonstrate:

- Use `.` to access a nested namespace within a namespace, as the following example of a using directive shows:

```
C#
using System.Collections.Generic;
```

- Use `.` to form a *qualified name* to access a type within a namespace, as the following code shows:

```
C#
System.Collections.Generic.IEnumerable<int> numbers = [1, 2, 3];
```

Use a using directive to make the use of qualified names optional.

- Use `.` to access type members, static and nonstatic, as the following code shows:

```
C#
```

```
List<double> constants =
[
    Math.PI,
    Math.E
];
Console.WriteLine($"{constants.Count} values to show:");
Console.WriteLine(string.Join(", ", constants));
// Output:
// 2 values to show:
// 3.14159265358979, 2.71828182845905
```

You can also use `.` to access an extension method.

Indexer operator []

Square brackets, `[]`, are typically used for array, indexer, or pointer element access. Beginning with C# 12, `[]` encloses a collection expression.

Array access

The following example demonstrates how to access array elements:

```
C#
```

```
int[] fib = new int[10];
fib[0] = fib[1] = 1;
for (int i = 2; i < fib.Length; i++)
{
    fib[i] = fib[i - 1] + fib[i - 2];
}
Console.WriteLine(fib[fib.Length - 1]);  // output: 55

double[,] matrix = new double[2,2];
matrix[0,0] = 1.0;
matrix[0,1] = 2.0;
matrix[1,0] = matrix[1,1] = 3.0;
var determinant = matrix[0,0] * matrix[1,1] - matrix[1,0] * matrix[0,1];
Console.WriteLine(determinant);  // output: -3
```

If an array index is outside the bounds of the corresponding dimension of an array, an IndexOutOfRangeException is thrown.

As the preceding example shows, you also use square brackets when you declare an array type or instantiate an array instance.

For more information about arrays, see Arrays.

Indexer access

The following example uses the .NET Dictionary<TKey,TValue> type to demonstrate indexer access:

C#

```
var dict = new Dictionary<string, double>();
dict["one"] = 1;
dict["pi"] = Math.PI;
Console.WriteLine(dict["one"] + dict["pi"]);  // output: 4.14159265358979
```

Indexers allow you to index instances of a user-defined type in the similar way as array indexing. Unlike array indices, which must be integer, the indexer parameters can be declared to be of any type.

For more information about indexers, see Indexers.

Other usages of []

For information about pointer element access, see the Pointer element access operator [] section of the Pointer related operators article. For information about collection expressions, see the collection expressions article.

You also use square brackets to specify attributes:

C#

```
[System.Diagnostics.Conditional("DEBUG")]
void TraceMethod() {}
```

Additionally, square brackets can be used to designate list patterns for use in pattern matching or testing.

C#

```
arr is ([1, 2, ..])
//Specifies that an array starts with (1, 2)
```

Null-conditional operators `?.` and `?[]`

A null-conditional operator applies a member access (`?.`) or element access (`?[]`) operation to its operand only if that operand evaluates to non-null; otherwise, it returns `null`. In other words:

- If `a` evaluates to `null`, the result of `a?.x` or `a?[x]` is `null`.
- If `a` evaluates to non-null, the result of `a?.x` or `a?[x]` is the same as the result of `a.x` or `a[x]`, respectively.

> **Note**
>
> If `a.x` or `a[x]` throws an exception, `a?.x` or `a?[x]` would throw the same exception for non-null `a`. For example, if `a` is a non-null array instance and `x` is outside the bounds of `a`, `a?[x]` would throw an IndexOutOfRangeException.

The null-conditional operators are short-circuiting. That is, if one operation in a chain of conditional member or element access operations returns `null`, the rest of the chain doesn't execute. In the following example, `B` isn't evaluated if `A` evaluates to `null` and `C` isn't evaluated if `A` or `B` evaluates to `null`:

C#

```
A?.B?.Do(C);
A?.B?[C];
```

If `A` might be null but `B` and `C` wouldn't be null if A isn't null, you only need to apply the null-conditional operator to `A`:

C#

```
A?.B.C();
```

In the preceding example, `B` isn't evaluated and `C()` isn't called if `A` is null. However, if the chained member access is interrupted, for example by parentheses as in `(A?.B).C()`, short-circuiting doesn't happen.

The following examples demonstrate the usage of the `?.` and `?[]` operators:

C#

```
double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
{
    return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
}

var sum1 = SumNumbers(null, 0);
Console.WriteLine(sum1);  // output: NaN

List<double[]?> numberSets =
[
    [1.0, 2.0, 3.0],
    null
];

var sum2 = SumNumbers(numberSets, 0);
Console.WriteLine(sum2);  // output: 6

var sum3 = SumNumbers(numberSets, 1);
Console.WriteLine(sum3);  // output: NaN
```

C#

```
namespace MemberAccessOperators2;

public static class NullConditionalShortCircuiting
{
    public static void Main()
    {
        Person? person = null;
        person?.Name.Write(); // no output: Write() is not called due to short-
circuit.
        try
        {
            (person?.Name).Write();
        }
        catch (NullReferenceException)
        {
            Console.WriteLine("NullReferenceException");
        }; // output: NullReferenceException
    }
}

public class Person
{
    public required FullName Name { get; set; }
}

public class FullName
{
    public required string FirstName { get; set; }
    public required string LastName { get; set; }
```

```
    public void Write() => Console.WriteLine($"{FirstName} {LastName}");
}
```

The first preceding example also uses the null-coalescing operator ?? to specify an alternative expression to evaluate in case the result of a null-conditional operation is `null`.

If `a.x` or `a[x]` is of a non-nullable value type `T`, `a?.x` or `a?[x]` is of the corresponding nullable value type `T?`. If you need an expression of type `T`, apply the null-coalescing operator `??` to a null-conditional expression, as the following example shows:

C#

```
int GetSumOfFirstTwoOrDefault(int[]? numbers)
{
    if ((numbers?.Length ?? 0) < 2)
    {
        return 0;
    }
    return numbers[0] + numbers[1];
}

Console.WriteLine(GetSumOfFirstTwoOrDefault(null));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault([]));  // output: 0
Console.WriteLine(GetSumOfFirstTwoOrDefault([3, 4, 5]));  // output: 7
```

In the preceding example, if you don't use the `??` operator, `numbers?.Length < 2` evaluates to `false` when `numbers` is `null`.

> **Note**
>
> The `?.` operator evaluates its left-hand operand no more than once, guaranteeing that it can't be changed to `null` after being verified as non-null.

Beginning in C# 14, assignment is permissible with a null conditional access expression (`?.` and `?[]`) on reference types. For example, see the following method:

C#

```
person?.FirstName = "Scott";
messages?[5] = "five";
```

The preceding example shows assignment to a property and an indexed element on a reference type that might be null. An important behavior for this assignment is that the expression on the right-hand side of the `=` is evaluated only when the left-hand side is known

to be non-null. For example, in the following code, the function `GenerateNextIndex` is called only when the `values` array isn't null. If the `values` array is null, `GenerateNextIndex` isn't called:

```
C#
```

```
person?.FirstName = "Scott";
messages?[5] = "five";
```

In other words, the preceding code is equivalent to the following code using an `if` statement for the null check:

```
C#
```

```
if (values is not null)
{
    values[2] = GenerateNextIndex();
}
```

In addition to assignment, any form of compound assignment, such as `+=` or `-=`, are allowed. However, increment (`++`) and decrement (`--`) aren't allowed.

This enhancement doesn't classify a null conditional expression as a variable. It can't be `ref` assigned, nor can it be assigned to a `ref` variable or passed to a method as a `ref` or `out` argument.

Thread-safe delegate invocation

Use the `?.` operator to check if a delegate is non-null and invoke it in a thread-safe way (for example, when you raise an event), as the following code shows:

```
C#
```

```
PropertyChanged?.Invoke(…)
```

That code is equivalent to the following code:

```
C#
```

```
var handler = this.PropertyChanged;
if (handler != null)
{
    handler(…);
}
```

The preceding example is a thread-safe way to ensure that only a non-null `handler` is invoked. Because delegate instances are immutable, no thread can change the object referenced by the `handler` local variable. In particular, if the code executed by another thread unsubscribes from the `PropertyChanged` event and `PropertyChanged` becomes `null` before `handler` is invoked, the object referenced by `handler` remains unaffected.

Invocation expression ()

Use parentheses, `()`, to call a method or invoke a delegate.

The following code demonstrates how to call a method, with or without arguments, and invoke a delegate:

```
C#

Action<int> display = s => Console.WriteLine(s);

List<int> numbers =
[
    10,
    17
];
display(numbers.Count);   // output: 2

numbers.Clear();
display(numbers.Count);   // output: 0
```

You also use parentheses when you invoke a constructor with the new operator.

Other usages of ()

You also use parentheses to adjust the order in which to evaluate operations in an expression. For more information, see C# operators.

Cast expressions, which perform explicit type conversions, also use parentheses.

Index from end operator ^

Index and range operators can be used with a type that is *countable*. A *countable* type is a type that has an `int` property named either `Count` or `Length` with an accessible `get` accessor. Collection expressions also rely on *countable* types.

> **Note**

Single dimensional arrays are *countable*. Multi-dimensional arrays aren't. The `^` and `..` (range) operators can't be used in multi-dimensional arrays.

The `^` operator indicates the element position from the end of a sequence. For a sequence of length `length`, `^n` points to the element with offset `length - n` from the start of a sequence. For example, `^1` points to the last element of a sequence and `^length` points to the first element of a sequence.

C#

```
int[] xs = [0, 10, 20, 30, 40];
int last = xs[^1];
Console.WriteLine(last);  // output: 40

List<string> lines = ["one", "two", "three", "four"];
string prelast = lines[^2];
Console.WriteLine(prelast);  // output: three

string word = "Twenty";
Index toFirst = ^word.Length;
char first = word[toFirst];
Console.WriteLine(first);  // output: T
```

As the preceding example shows, expression `^e` is of the System.Index type. In expression `^e`, the result of `e` must be implicitly convertible to `int`.

You can also use the `^` operator with the range operator to create a range of indices. For more information, see Indices and ranges.

Beginning with C# 13, the Index from the end operator can be used in an object initializer.

Range operator `..`

The `..` operator specifies the start and end of a range of indices as its operands. The left-hand operand is an *inclusive* start of a range. The right-hand operand is an *exclusive* end of a range. Either of the operands can be an index from the start or from the end of a sequence, as the following example shows:

C#

```
int[] numbers = [0, 10, 20, 30, 40, 50];
int start = 1;
int amountToTake = 3;
int[] subset = numbers[start..(start + amountToTake)];
Display(subset);  // output: 10 20 30
```

```
int margin = 1;
int[] inner = numbers[margin..^margin];
Display(inner);  // output: 10 20 30 40

string line = "one two three";
int amountToTakeFromEnd = 5;
Range endIndices = ^amountToTakeFromEnd..^0;
string end = line[endIndices];
Console.WriteLine(end);  // output: three

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

As the preceding example shows, expression `a..b` is of the System.Range type. In expression `a..b`, the results of `a` and `b` must be implicitly convertible to Int32 or Index.

> **Important**
>
> Implicit conversions from `int` to `Index` throw an ArgumentOutOfRangeException when the value is negative.

You can omit any of the operands of the `..` operator to obtain an open-ended range:

- `a..` is equivalent to `a..^0`
- `..b` is equivalent to `0..b`
- `..` is equivalent to `0..^0`

C#

```
int[] numbers = [0, 10, 20, 30, 40, 50];
int amountToDrop = numbers.Length / 2;

int[] rightHalf = numbers[amountToDrop..];
Display(rightHalf);  // output: 30 40 50

int[] leftHalf = numbers[..^amountToDrop];
Display(leftHalf);  // output: 0 10 20

int[] all = numbers[..];
Display(all);  // output: 0 10 20 30 40 50

void Display<T>(IEnumerable<T> xs) => Console.WriteLine(string.Join(" ", xs));
```

The following table shows various ways to express collection ranges:

Expand table

Range operator expression	Description
`..`	All values in the collection.
`..end`	Values from the start to the `end` exclusively.
`start..`	Values from the `start` inclusively to the end.
`start..end`	Values from the `start` inclusively to the `end` exclusively.
`^start..`	Values from the `start` inclusively to the end counting from the end.
`..^end`	Values from the start to the `end` exclusively counting from the end.
`start..^end`	Values from `start` inclusively to `end` exclusively counting from the end.
`^start..^end`	Values from `start` inclusively to `end` exclusively both counting from the end.

The following example demonstrates the effect of using all the ranges presented in the preceding table:

C#

```
int[] oneThroughTen =
[
    1, 2, 3, 4, 5, 6, 7, 8, 9, 10
];

Write(oneThroughTen, ..);
Write(oneThroughTen, ..3);
Write(oneThroughTen, 2..);
Write(oneThroughTen, 3..5);
Write(oneThroughTen, ^2..);
Write(oneThroughTen, ..^3);
Write(oneThroughTen, 3..^4);
Write(oneThroughTen, ^4..^2);

static void Write(int[] values, Range range) =>
    Console.WriteLine($"{range}:\t{string.Join(", ", values[range])}");
// Sample output:
//      0..^0:      1, 2, 3, 4, 5, 6, 7, 8, 9, 10
//      0..3:       1, 2, 3
//      2..^0:      3, 4, 5, 6, 7, 8, 9, 10
//      3..5:       4, 5
//      ^2..^0:     9, 10
//      0..^3:      1, 2, 3, 4, 5, 6, 7
//      3..^4:      4, 5, 6
//      ^4..^2:     7, 8
```

For more information, see Indices and ranges.

The `..` token is also used for the spread element in a collection expression.

Operator overloadability

The `.`, `()`, `^`, and `..` operators can't be overloaded. The `[]` operator is also considered a non-overloadable operator. Use indexers to support indexing with user-defined types.

C# language specification

For more information, see the following sections of the C# language specification:

- Member access
- Element access
- Null-conditional member access
- Invocation expressions

For more information about indices and ranges, see the feature proposal note.

See also

- Use index operator (style rule IDE0056)
- Use range operator (style rule IDE0057)
- Use conditional delegate call (style rule IDE1005)
- C# operators and expressions
- ?? (null-coalescing operator)
- :: operator

Type-testing operators and cast expressions - `is`, `as`, `typeof`, and casts

Article • 02/19/2025

These operators and expressions perform type checking or type conversion. The `is` operator checks if the run-time type of an expression is compatible with a given type. The `as` operator explicitly converts an expression to a given type if its run-time type is compatible with that type. Cast expressions perform an explicit conversion to a target type. The `typeof` operator obtains the System.Type instance for a type.

The `is` operator

The `is` operator checks if the run-time type of an expression result is compatible with a given type. The `is` operator also tests an expression result against a pattern.

The expression with the type-testing `is` operator has the following form

```
C#

E is T
```

Where `E` is an expression that returns a value and `T` is the name of a type or a type parameter. `E` can't be an anonymous method or a lambda expression.

The `is` operator returns `true` when an expression result is non-null and any of the following conditions are true:

- The run-time type of an expression result is `T`.
- The run-time type of an expression result derives from type `T`, implements interface `T`, or another implicit reference conversion exists from it to `T`.
- The run-time type of an expression result is a nullable value type with the underlying type `T` and the Nullable<T>.HasValue is `true`.
- A boxing or unboxing conversion exists from the run-time type of an expression result to type `T` when the expression isn't an instance of a `ref struct`.

The `is` operator doesn't consider user-defined conversions or implicit span conversions.

The following example demonstrates that the `is` operator returns `true` if the run-time type of an expression result derives from a given type, that is, there exists a reference conversion between types:

```
C#
```

```
public class Base { }

public class Derived : Base { }

public static class IsOperatorExample
{
    public static void Main()
    {
        object b = new Base();
        Console.WriteLine(b is Base);  // output: True
        Console.WriteLine(b is Derived);  // output: False

        object d = new Derived();
        Console.WriteLine(d is Base);  // output: True
        Console.WriteLine(d is Derived); // output: True
    }
}
```

The next example shows that the `is` operator takes into account boxing and unboxing conversions but doesn't consider numeric conversions:

```
C#
```

```
int i = 27;
Console.WriteLine(i is System.IFormattable);  // output: True

object iBoxed = i;
Console.WriteLine(iBoxed is int);  // output: True
Console.WriteLine(iBoxed is long);  // output: False
```

For information about C# conversions, see the Conversions chapter of the C# language specification.

Type testing with pattern matching

The `is` operator also tests an expression result against a pattern. The following example shows how to use a declaration pattern to check the run-time type of an expression:

```
C#
```

```
int i = 23;
object iBoxed = i;
int? jNullable = 7;
if (iBoxed is int a && jNullable is int b)
{
    Console.WriteLine(a + b);  // output 30
}
```

For information about the supported patterns, see Patterns.

The `as` operator

The `as` operator explicitly converts the result of an expression to a given reference or nullable value type. If the conversion isn't possible, the `as` operator returns `null`. Unlike a cast expression, the `as` operator never throws an exception.

The expression of the form

C#

```
E as T
```

Where `E` is an expression that returns a value and `T` is the name of a type or a type parameter, produces the same result as

C#

```
E is T ? (T)(E) : (T)null
```

Except that `E` is only evaluated once.

The `as` operator considers only reference, nullable, boxing, and unboxing conversions. You can't use the `as` operator to perform a user-defined conversion. To do that, use a cast expression.

The following example demonstrates the usage of the `as` operator:

C#

```
IEnumerable<int> numbers = new List<int>(){10, 20, 30};
IList<int> indexable = numbers as IList<int>;
if (indexable != null)
{
    Console.WriteLine(indexable[0] + indexable[indexable.Count - 1]);  //
```

```
output: 40
}
```

> **Note**
>
> As the preceding example shows, you need to compare the result of the `as` expression with `null` to check if the conversion was successful. You can use the **is operator** both to test if the conversion succeeds and, if it succeeds, assign its result to a new variable.

Cast expression

A cast expression of the form `(T)E` performs an explicit conversion of the result of expression `E` to type `T`. If no explicit conversion exists from the type of `E` to type `T`, a compile-time error occurs. At run time, an explicit conversion might not succeed and a cast expression might throw an exception.

The following example demonstrates explicit numeric and reference conversions:

C#

```
double x = 1234.7;
int a = (int)x;
Console.WriteLine(a);   // output: 1234

int[] ints = [10, 20, 30];
IEnumerable<int> numbers = ints;
IList<int> list = (IList<int>)numbers;
Console.WriteLine(list.Count);  // output: 3
Console.WriteLine(list[1]);  // output: 20
```

For information about supported explicit conversions, see the Explicit conversions section of the C# language specification. For information about how to define a custom explicit or implicit type conversion, see User-defined conversion operators.

Other usages of ()

You also use parentheses to call a method or invoke a delegate.

Other use of parentheses is to adjust the order in which to evaluate operations in an expression. For more information, see C# operators.

The `typeof` operator

The `typeof` operator obtains the System.Type instance for a type. The argument to the `typeof` operator must be the name of a type or a type parameter, as the following example shows:

C#

```
void PrintType<T>() => Console.WriteLine(typeof(T));

Console.WriteLine(typeof(List<string>));
PrintType<int>();
PrintType<System.Int32>();
PrintType<Dictionary<int, char>>();
// Output:
// System.Collections.Generic.List`1[System.String]
// System.Int32
// System.Int32
// System.Collections.Generic.Dictionary`2[System.Int32,System.Char]
```

The argument mustn't be a type that requires metadata annotations. Examples include the following types:

- `dynamic`
- `string?` (or any nullable reference type)

These types aren't directly represented in metadata. The types include attributes that describe the underlying type. In both cases, you can use the underlying type. Instead of `dynamic`, you can use `object`. Instead of `string?`, you can use `string`.

You can also use the `typeof` operator with unbound generic types. The name of an unbound generic type must contain the appropriate number of commas, which is one less than the number of type parameters. The following example shows the usage of the `typeof` operator with an unbound generic type:

C#

```
Console.WriteLine(typeof(Dictionary<,>));
// Output:
// System.Collections.Generic.Dictionary`2[TKey,TValue]
```

An expression can't be an argument of the `typeof` operator. To get the System.Type instance for the run-time type of an expression result, use the Object.GetType method.

Type testing with the `typeof` operator

Use the `typeof` operator to check if the run-time type of the expression result exactly matches a given type. The following example demonstrates the difference between type checking done with the `typeof` operator and the is operator:

```
C#
```

```
public class Animal { }

public class Giraffe : Animal { }

public static class TypeOfExample
{
    public static void Main()
    {
        object b = new Giraffe();
        Console.WriteLine(b is Animal);  // output: True
        Console.WriteLine(b.GetType() == typeof(Animal));  // output: False

        Console.WriteLine(b is Giraffe);  // output: True
        Console.WriteLine(b.GetType() == typeof(Giraffe));  // output: True
    }
}
```

Operator overloadability

The `is`, `as`, and `typeof` operators can't be overloaded.

A user-defined type can't overload the `()` operator, but can define custom type conversions performed by a cast expression. For more information, see User-defined conversion operators.

C# language specification

For more information, see the following sections of the C# language specification:

- The is operator
- The as operator
- Cast expressions
- The typeof operator

See also

- C# operators and expressions
- How to safely cast by using pattern matching and the is and as operators
- Generics in .NET

User-defined explicit and implicit conversion operators

Article • 02/25/2025

A user-defined type can define a custom implicit or explicit conversion from or to another type, provided a standard conversion doesn't exist between the same two types. Implicit conversions don't require special syntax to be invoked and can occur in various situations, for example, in assignments and methods invocations. Predefined C# implicit conversions always succeed and never throw an exception. User-defined implicit conversions should behave in that way as well. If a custom conversion can throw an exception or lose information, define it as an explicit conversion.

The is and as operators don't consider user-defined conversions. Use a cast expression to invoke a user-defined explicit conversion.

Use the `operator` and `implicit` or `explicit` keywords to define an implicit or explicit conversion, respectively. The type that defines a conversion must be either a source type or a target type of that conversion. A conversion between two user-defined types can be defined in either of the two types.

The following example demonstrates how to define an implicit and explicit conversion:

C#

```
using System;

public readonly struct Digit
{
    private readonly byte digit;

    public Digit(byte digit)
    {
        if (digit > 9)
        {
            throw new ArgumentOutOfRangeException(nameof(digit), "Digit
cannot be greater than nine.");
        }
        this.digit = digit;
    }

    public static implicit operator byte(Digit d) => d.digit;
    public static explicit operator Digit(byte b) => new Digit(b);

    public override string ToString() => $"{digit}";
}
```

```
public static class UserDefinedConversions
{
    public static void Main()
    {
        var d = new Digit(7);

        byte number = d;
        Console.WriteLine(number);  // output: 7

        Digit digit = (Digit)number;
        Console.WriteLine(digit);  // output: 7
    }
}
```

Beginning with C# 11, you can define *checked* explicit conversion operators. For more information, see the User-defined checked operators section of the Arithmetic operators article.

You also use the `operator` keyword to overload a predefined C# operator. For more information, see Operator overloading.

C# language specification

For more information, see the following sections of the C# language specification:

- Conversion operators
- User-defined conversions
- Implicit conversions
- Explicit conversions

See also

- C# operators and expressions
- Operator overloading
- Type-testing and cast operators
- Casting and type conversion
- Design guidelines - Conversion operators
- Chained user-defined explicit conversions in C#

Pointer related operators - take the address of variables, dereference storage locations, and access memory locations

Article • 04/12/2023

The pointer operators enable you to take the address of a variable (`&`), dereference a pointer (`*`), compare pointer values, and add or subtract pointers and integers.

You use the following operators to work with pointers:

- Unary & (address-of) operator: to get the address of a variable
- Unary * (pointer indirection) operator: to obtain the variable pointed by a pointer
- The -> (member access) and [] (element access) operators
- Arithmetic operators +, -, ++, and --
- Comparison operators ==, !=, <, >, <=, and >=

For information about pointer types, see Pointer types.

> **Note**
>
> Any operation with pointers requires an unsafe context. The code that contains unsafe blocks must be compiled with the AllowUnsafeBlocks compiler option.

Address-of operator &

The unary `&` operator returns the address of its operand:

C#

```
unsafe
{
    int number = 27;
    int* pointerToNumber = &number;

    Console.WriteLine($"Value of the variable: {number}");
    Console.WriteLine($"Address of the variable:
{(long)pointerToNumber:X}");
}
// Output is similar to:
```

```
// Value of the variable: 27
// Address of the variable: 6C1457DBD4
```

The operand of the `&` operator must be a fixed variable. *Fixed* variables are variables that reside in storage locations that are unaffected by operation of the garbage collector. In the preceding example, the local variable `number` is a fixed variable, because it resides on the stack. Variables that reside in storage locations that can be affected by the garbage collector (for example, relocated) are called *movable* variables. Object fields and array elements are examples of movable variables. You can get the address of a movable variable if you "fix", or "pin", it with a fixed statement. The obtained address is valid only inside the block of a `fixed` statement. The following example shows how to use a `fixed` statement and the `&` operator:

C#

```
unsafe
{
    byte[] bytes = { 1, 2, 3 };
    fixed (byte* pointerToFirst = &bytes[0])
    {
        // The address stored in pointerToFirst
        // is valid only inside this fixed statement block.
    }
}
```

You can't get the address of a constant or a value.

For more information about fixed and movable variables, see the Fixed and moveable variables section of the C# language specification.

The binary `&` operator computes the logical AND of its Boolean operands or the bitwise logical AND of its integral operands.

Pointer indirection operator *

The unary pointer indirection operator `*` obtains the variable to which its operand points. It's also known as the dereference operator. The operand of the `*` operator must be of a pointer type.

C#

```
unsafe
{
    char letter = 'A';
    char* pointerToLetter = &letter;
```

```
    Console.WriteLine($"Value of the `letter` variable: {letter}");
    Console.WriteLine($"Address of the `letter` variable:
{(long)pointerToLetter:X}");

    *pointerToLetter = 'Z';
    Console.WriteLine($"Value of the `letter` variable after update:
{letter}");
}
// Output is similar to:
// Value of the `letter` variable: A
// Address of the `letter` variable: DCB977DDF4
// Value of the `letter` variable after update: Z
```

You can't apply the `*` operator to an expression of type `void*`.

The binary `*` operator computes the product of its numeric operands.

Pointer member access operator ->

The `->` operator combines pointer indirection and member access. That is, if `x` is a pointer of type `T*` and `y` is an accessible member of type `T`, an expression of the form

C#

```
x->y
```

is equivalent to

C#

```
(*x).y
```

The following example demonstrates the usage of the `->` operator:

C#

```
public struct Coords
{
    public int X;
    public int Y;
    public override string ToString() => $"({X}, {Y})";
}

public class PointerMemberAccessExample
{
    public static unsafe void Main()
    {
```

```
        Coords coords;
        Coords* p = &coords;
        p->X = 3;
        p->Y = 4;
        Console.WriteLine(p->ToString());  // output: (3, 4)
    }
}
```

You can't apply the `->` operator to an expression of type `void*`.

Pointer element access operator []

For an expression `p` of a pointer type, a pointer element access of the form `p[n]` is evaluated as `*(p + n)`, where `n` must be of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`. For information about the behavior of the `+` operator with pointers, see the Addition or subtraction of an integral value to or from a pointer section.

The following example demonstrates how to access array elements with a pointer and the `[]` operator:

C#

```
unsafe
{
    char* pointerToChars = stackalloc char[123];

    for (int i = 65; i < 123; i++)
    {
        pointerToChars[i] = (char)i;
    }

    Console.Write("Uppercase letters: ");
    for (int i = 65; i < 91; i++)
    {
        Console.Write(pointerToChars[i]);
    }
}
// Output:
// Uppercase letters: ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

In the preceding example, a stackalloc expression allocates a block of memory on the stack.

> **Note**
>
> The pointer element access operator doesn't check for out-of-bounds errors.

You can't use `[]` for pointer element access with an expression of type `void*`.

You can also use the `[]` operator for array element or indexer access.

Pointer arithmetic operators

You can perform the following arithmetic operations with pointers:

- Add or subtract an integral value to or from a pointer
- Subtract two pointers
- Increment or decrement a pointer

You can't perform those operations with pointers of type `void*`.

For information about supported arithmetic operations with numeric types, see Arithmetic operators.

Addition or subtraction of an integral value to or from a pointer

For a pointer `p` of type `T*` and an expression `n` of a type implicitly convertible to `int`, `uint`, `long`, or `ulong`, addition and subtraction are defined as follows:

- Both `p + n` and `n + p` expressions produce a pointer of type `T*` that results from adding `n * sizeof(T)` to the address given by `p`.
- The `p - n` expression produces a pointer of type `T*` that results from subtracting `n * sizeof(T)` from the address given by `p`.

The sizeof operator obtains the size of a type in bytes.

The following example demonstrates the usage of the `+` operator with a pointer:

C#

```
unsafe
{
    const int Count = 3;
    int[] numbers = new int[Count] { 10, 20, 30 };
    fixed (int* pointerToFirst = &numbers[0])
    {
        int* pointerToLast = pointerToFirst + (Count - 1);

        Console.WriteLine($"Value {*pointerToFirst} at address
{(long)pointerToFirst}");
        Console.WriteLine($"Value {*pointerToLast} at address
```

```
{(long)pointerToLast}");
    }
}
// Output is similar to:
// Value 10 at address 1818345918136
// Value 30 at address 1818345918144
```

Pointer subtraction

For two pointers `p1` and `p2` of type `T*`, the expression `p1 - p2` produces the difference between the addresses given by `p1` and `p2` divided by `sizeof(T)`. The type of the result is `long`. That is, `p1 - p2` is computed as `((long)(p1) - (long)(p2)) / sizeof(T)`.

The following example demonstrates the pointer subtraction:

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2, 3, 4, 5 };
    int* p1 = &numbers[1];
    int* p2 = &numbers[5];
    Console.WriteLine(p2 - p1);  // output: 4
}
```

Pointer increment and decrement

The `++` increment operator adds 1 to its pointer operand. The `--` decrement operator subtracts 1 from its pointer operand.

Both operators are supported in two forms: postfix (`p++` and `p--`) and prefix (`++p` and `--p`). The result of `p++` and `p--` is the value of `p` *before* the operation. The result of `++p` and `--p` is the value of `p` *after* the operation.

The following example demonstrates the behavior of both postfix and prefix increment operators:

C#

```
unsafe
{
    int* numbers = stackalloc int[] { 0, 1, 2 };
    int* p1 = &numbers[0];
    int* p2 = p1;
    Console.WriteLine($"Before operation: p1 - {(long)p1}, p2 -
{(long)p2}");
```

```
    Console.WriteLine($"Postfix increment of p1: {(long)(p1++)}");
    Console.WriteLine($"Prefix increment of p2: {(long)(++p2)}");
    Console.WriteLine($"After operation: p1 - {(long)p1}, p2 - {(long)p2}");
}
// Output is similar to
// Before operation: p1 - 816489946512, p2 - 816489946512
// Postfix increment of p1: 816489946512
// Prefix increment of p2: 816489946516
// After operation: p1 - 816489946516, p2 - 816489946516
```

Pointer comparison operators

You can use the `==`, `!=`, `<`, `>`, `<=`, and `>=` operators to compare operands of any pointer type, including `void*`. Those operators compare the addresses given by the two operands as if they're unsigned integers.

For information about the behavior of those operators for operands of other types, see the Equality operators and Comparison operators articles.

Operator precedence

The following list orders pointer related operators starting from the highest precedence to the lowest:

- Postfix increment `x++` and decrement `x--` operators and the `->` and `[]` operators
- Prefix increment `++x` and decrement `--x` operators and the `&` and `*` operators
- Additive `+` and `-` operators
- Comparison `<`, `>`, `<=`, and `>=` operators
- Equality `==` and `!=` operators

Use parentheses, `()`, to change the order of evaluation imposed by operator precedence.

For the complete list of C# operators ordered by precedence level, see the Operator precedence section of the C# operators article.

Operator overloadability

A user-defined type can't overload the pointer related operators `&`, `*`, `->`, and `[]`.

C# language specification

For more information, see the following sections of the C# language specification:

- Fixed and moveable variables
- The address-of operator
- Pointer indirection
- Pointer member access
- Pointer element access
- Pointer arithmetic
- Pointer increment and decrement
- Pointer comparison

See also

- C# operators and expressions
- Unsafe code, pointer types, and function pointers
- unsafe keyword
- fixed statement
- stackalloc expression
- sizeof operator

Assignment operators (C# reference)

06/13/2025

The assignment operator `=` assigns the *value* of its right-hand operand to a variable, a property, or an indexer element given by its left-hand operand. The result of an assignment expression is the value assigned to the left-hand operand. The type of the right-hand operand must be the same as the type of the left-hand operand or implicitly convertible to it.

The assignment operator `=` is right-associative, that is, an expression of the form

C#

```
a = b = c
```

Is evaluated as

C#

```
a = (b = c)
```

The following example demonstrates the usage of the assignment operator with a local variable, a property, and an indexer element as its left-hand operand:

C#

```
List<double> numbers = [1.0, 2.0, 3.0];

Console.WriteLine(numbers.Capacity);
numbers.Capacity = 100;
Console.WriteLine(numbers.Capacity);
// Output:
// 4
// 100

int newFirstElement;
double originalFirstElement = numbers[0];
newFirstElement = 5;
numbers[0] = newFirstElement;
Console.WriteLine(originalFirstElement);
Console.WriteLine(numbers[0]);
// Output:
// 1
// 5
```

The left-hand operand of an assignment receives the *value* of the right-hand operand. When the operands are of value types, assignment copies the contents of the right-hand operand.

When the operands are of reference types, assignment copies the reference to the object.

This operation is called *value assignment*: the value is assigned.

Beginning with C# 14, the left hand side of a value assignment can include a null conditional member expression, such as `?.` or `?[]`. If the left hand side is null, the right hand side expression isn't evaluated.

ref assignment

Ref assignment `= ref` makes its left-hand operand an alias to the right-hand operand, as the following example demonstrates:

C#

```
void Display(double[] s) => Console.WriteLine(string.Join(" ", s));

double[] arr = { 0.0, 0.0, 0.0 };
Display(arr);

ref double arrayElement = ref arr[0];
arrayElement = 3.0;
Display(arr);

arrayElement = ref arr[arr.Length - 1];
arrayElement = 5.0;
Display(arr);
// Output:
// 0 0 0
// 3 0 0
// 3 0 5
```

In the preceding example, the local reference variable `arrayElement` is initialized as an alias to the first array element. Then, it's `ref` reassigned to refer to the last array element. As it's an alias, when you update its value with an ordinary assignment operator `=`, the corresponding array element is also updated.

The left-hand operand of `ref` assignment can be a local reference variable, a ref field, and a ref, out, or in method parameter. Both operands must be of the same type.

A `ref` assignment means that a reference variable has a different referrent. It's no longer referring to its previous referrent. Using `ref =` on a `ref` parameter means the parameter no longer refers to its argument. Any actions that modify the state of the object after ref reassigning it make those modifications to the new item. For example, consider the following method:

```C#
private static void RefReassignAndModify(scoped ref string s)
{
    string sLocal = "Hello";
    Console.WriteLine(sLocal);  // Output: Hello

    s = ref sLocal;
    s = "World";
    Console.WriteLine(s);  // Output: World
```

The following usage shows that the assignment to the parameter `s` isn't visible after the method call because `s` was `ref` reassigned to refer to `sLocal` before the string was modified:

```C#
string msg = "Hi";
RefReassignAndModify(ref msg);
Console.WriteLine(msg); // Output: Hi!
```

Compound assignment

For a binary operator `op`, a compound assignment expression of the form

```C#
x op= y
```

Is equivalent to

```C#
x = x op y
```

Except that `x` is only evaluated once.

The arithmetic, Boolean logical, and bitwise logical and shift operators all support compound assignment.

Null-coalescing assignment

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

information, see the ?? and ??= operators article.

Operator overloadability

A user-defined type can't overload the assignment operator. However, a user-defined type can define an implicit conversion to another type. That way, the value of a user-defined type can be assigned to a variable, a property, or an indexer element of another type. For more information, see User-defined conversion operators.

If a user-defined type overloads a binary operator `op`, the `op=` operator, if it exists, is also implicitly overloaded. Beginning with C# 14, a user-defined type can explicitly overload the compound assignment operators (`op=`) to provide a more efficient implementation. Typically, a type overloads these operators because the value can be updated in place, rather than allocating a new instance to hold the result of the binary operation. If a type doesn't provide an explicit overload, the compiler generates the implicit overload.

C# language specification

For more information, see the Assignment operators section of the C# language specification and the User defined compound assignment feature specification.

See also

- C# operators and expressions
- ref keyword
- Use compound assignment (style rules IDE0054 and IDE0074)

Lambda expressions and anonymous functions

Article • 02/19/2025

You use a *lambda expression* to create an anonymous function. Use the lambda declaration operator `=>` to separate the lambda's parameter list from its body. A lambda expression can be of any of the following two forms:

- Expression lambda that has an expression as its body:

  ```
  C#

  (input-parameters) => expression
  ```

- Statement lambda that has a statement block as its body:

  ```
  C#

  (input-parameters) => { <sequence-of-statements> }
  ```

To create a lambda expression, you specify input parameters (if any) on the left side of the lambda operator and an expression or a statement block on the other side.

Any lambda expression can be converted to a delegate type. The types of its parameters and return value define the delegate type to which a lambda expression can be converted. If a lambda expression doesn't return a value, it can be converted to one of the `Action` delegate types; otherwise, it can be converted to one of the `Func` delegate types. For example, a lambda expression that has two parameters and returns no value can be converted to an Action<T1,T2> delegate. A lambda expression that has one parameter and returns a value can be converted to a Func<T,TResult> delegate. In the following example, the lambda expression `x => x * x`, which specifies a parameter named `x` and returns the value of `x` squared, is assigned to a variable of a delegate type:

```
C#

Func<int, int> square = x => x * x;
Console.WriteLine(square(5));
// Output:
// 25
```

Expression lambdas can also be converted to the expression tree types, as the following example shows:

```
C#
```

```
System.Linq.Expressions.Expression<Func<int, int>> e = x => x * x;
Console.WriteLine(e);
// Output:
// x => (x * x)
```

You use lambda expressions in any code that requires instances of delegate types or expression trees. One example is the argument to the Task.Run(Action) method to pass the code that should be executed in the background. You can also use lambda expressions when you write LINQ in C#, as the following example shows:

```
C#
```

```
int[] numbers = { 2, 3, 4, 5 };
var squaredNumbers = numbers.Select(x => x * x);
Console.WriteLine(string.Join(" ", squaredNumbers));
// Output:
// 4 9 16 25
```

When you use method-based syntax to call the Enumerable.Select method in the System.Linq.Enumerable class, for example in LINQ to Objects and LINQ to XML, the parameter is a delegate type System.Func<T,TResult>. When you call the Queryable.Select method in the System.Linq.Queryable class, for example in LINQ to SQL, the parameter type is an expression tree type Expression<Func<TSource,TResult>>. In both cases, you can use the same lambda expression to specify the parameter value. That makes the two `Select` calls to look similar although in fact the type of objects created from the lambdas is different.

Expression lambdas

A lambda expression with an expression on the right side of the `=>` operator is called an *expression lambda*. An expression lambda returns the result of the expression and takes the following basic form:

```
C#
```

```
(input-parameters) => expression
```

The body of an expression lambda can consist of a method call. However, when creating expression trees evaluated by a query provider, limit method calls to those methods recognized by the query provider. Otherwise, the query provider can't replicate the method's function.

Statement lambdas

A statement lambda resembles an expression lambda except that its statements are enclosed in braces:

```
C#

(input-parameters) => { <sequence-of-statements> }
```

The body of a statement lambda can consist of any number of statements; however, in practice there are typically no more than two or three.

```
C#

Action<string> greet = name =>
{
    string greeting = $"Hello {name}!";
    Console.WriteLine(greeting);
};
greet("World");
// Output:
// Hello World!
```

You can't use statement lambdas to create expression trees.

Input parameters of a lambda expression

You enclose input parameters of a lambda expression in parentheses. Specify zero input parameters with empty parentheses:

```
C#

Action line = () => Console.WriteLine();
```

If a lambda expression has only one input parameter, parentheses are optional:

```
C#
```

```
Func<double, double> cube = x => x * x * x;
```

Two or more input parameters are separated by commas:

C#

```
Func<int, int, bool> testForEquality = (x, y) => x == y;
```

The compiler typically infers the types for parameters to lambda expressions, referred to as an *implicitly typed parameter list*. You can specify the types explicitly, referred to as an *explicitly typed parameter list*. An explicitly typed parameter list is shown in the following example. :

C#

```
Func<int, string, bool> isTooLong = (int x, string s) => s.Length > x;
```

Input parameter types must be all explicit or all implicit; otherwise, a CS0748 compiler error occurs. Before C# 14, you must include the explicit type on a parameter if it has any modifiers, such as `ref` or `out`. In C# 14, that restriction is removed. However, you must still declare the type if you use the `params` modifier.

You can use discards to specify two or more input parameters of a lambda expression that aren't used in the expression:

C#

```
Func<int, int, int> constant = (_, _) => 42;
```

Lambda discard parameters can be useful when you use a lambda expression to provide an event handler.

> **Note**
>
> For backwards compatibility, if only a single input parameter is named `_`, `_` is treated as the name of that parameter within that lambda expression.

Beginning with C# 12, you can provide *default values* for explicitly typed parameter lists. The syntax and the restrictions on default parameter values are the same as for methods and local functions. The following example declares a lambda expression with a default parameter, then calls it once using the default and once with two explicit parameters:

C#

```
var IncrementBy = (int source, int increment = 1) => source + increment;

Console.WriteLine(IncrementBy(5)); // 6
Console.WriteLine(IncrementBy(5, 2)); // 7
```

You can also declare lambda expressions with `params` arrays or collections as the last parameter in an explicitly typed parameter list:

C#

```
var sum = (params IEnumerable<int> values) =>
{
    int sum = 0;
    foreach (var value in values)
        sum += value;

    return sum;
};

var empty = sum();
Console.WriteLine(empty); // 0

var sequence = new[] { 1, 2, 3, 4, 5 };
var total = sum(sequence);
Console.WriteLine(total); // 15
```

As part of these updates, when a method group that has a default parameter is assigned to a lambda expression, that lambda expression also has the same default parameter. A method group with a `params` collection parameter can also be assigned to a lambda expression.

Lambda expressions with default parameters or `params` collections as parameters don't have natural types that correspond to `Func<>` or `Action<>` types. However, you can define delegate types that include default parameter values:

C#

```
delegate int IncrementByDelegate(int source, int increment = 1);
delegate int SumDelegate(params int[] values);
delegate int SumCollectionDelegate(params IEnumerable<int> values);
```

Or, you can use implicitly typed variables with `var` declarations to define the delegate type. The compiler synthesizes the correct delegate type.

For more information about default parameters on lambda expressions, see the feature spec for default parameters on lambda expressions.

Async lambdas

You can easily create lambda expressions and statements that incorporate asynchronous processing by using the async and await keywords. For example, the following Windows Forms example contains an event handler that calls and awaits an async method, `ExampleMethodAsync`.

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += button1_Click;
    }

    private async void button1_Click(object sender, EventArgs e)
    {
        await ExampleMethodAsync();
        textBox1.Text += "\r\nControl returned to Click event handler.\n";
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

You can add the same event handler by using an async lambda. To add this handler, add an `async` modifier before the lambda parameter list, as the following example shows:

C#

```
public partial class Form1 : Form
{
    public Form1()
    {
        InitializeComponent();
        button1.Click += async (sender, e) =>
        {
            await ExampleMethodAsync();
            textBox1.Text += "\r\nControl returned to Click event
handler.\n";
```

```
        };
    }

    private async Task ExampleMethodAsync()
    {
        // The following line simulates a task-returning asynchronous
process.
        await Task.Delay(1000);
    }
}
```

For more information about how to create and use async methods, see Asynchronous Programming with async and await.

Lambda expressions and tuples

The C# language provides built-in support for tuples. You can provide a tuple as an argument to a lambda expression, and your lambda expression can also return a tuple. In some cases, the C# compiler uses type inference to determine the types of tuple elements.

You define a tuple by enclosing a comma-delimited list of its components in parentheses. The following example uses tuple with three components to pass a sequence of numbers to a lambda expression, which doubles each value and returns a tuple with three components that contains the result of the multiplications.

C#

```
Func<(int, int, int), (int, int, int)> doubleThem = ns => (2 * ns.Item1, 2 *
ns.Item2, 2 * ns.Item3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
// Output:
// The set (2, 3, 4) doubled: (4, 6, 8)
```

Ordinarily, the fields of a tuple are named `Item1`, `Item2`, and so on. You can, however, define a tuple with named components, as the following example does.

C#

```
Func<(int n1, int n2, int n3), (int, int, int)> doubleThem = ns => (2 *
ns.n1, 2 * ns.n2, 2 * ns.n3);
var numbers = (2, 3, 4);
var doubledNumbers = doubleThem(numbers);
Console.WriteLine($"The set {numbers} doubled: {doubledNumbers}");
```

For more information about C# tuples, see Tuple types.

Lambdas with the standard query operators

LINQ to Objects, among other implementations, has an input parameter whose type is one of the Func<TResult> family of generic delegates. These delegates use type parameters to define the number and type of input parameters, and the return type of the delegate. `Func` delegates are useful for encapsulating user-defined expressions that are applied to each element in a set of source data. For example, consider the Func<T,TResult> delegate type:

C#

```
public delegate TResult Func<in T, out TResult>(T arg)
```

The delegate can be instantiated as a `Func<int, bool>` instance where `int` is an input parameter and `bool` is the return value. The return value is always specified in the last type parameter. For example, `Func<int, string, bool>` defines a delegate with two input parameters, `int` and `string`, and a return type of `bool`. The following `Func` delegate, when invoked, returns Boolean value that indicates whether the input parameter is equal to five:

C#

```
Func<int, bool> equalsFive = x => x == 5;
bool result = equalsFive(4);
Console.WriteLine(result);   // False
```

You can also supply a lambda expression when the argument type is an Expression<TDelegate>, for example in the standard query operators that are defined in the Queryable type. When you specify an Expression<TDelegate> argument, the lambda is compiled to an expression tree.

The following example uses the Count standard query operator:

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
int oddNumbers = numbers.Count(n => n % 2 == 1);
Console.WriteLine($"There are {oddNumbers} odd numbers in {string.Join(" ",
numbers)}");
```

The compiler can infer the type of the input parameter, or you can also specify it explicitly. This particular lambda expression counts those integers (`n`) which when divided by two have a remainder of 1.

The following example produces a sequence that contains all elements in the `numbers` array that precede the 9, because that's the first number in the sequence that doesn't meet the condition:

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstNumbersLessThanSix = numbers.TakeWhile(n => n < 6);
Console.WriteLine(string.Join(" ", firstNumbersLessThanSix));
// Output:
// 5 4 1 3
```

The following example specifies multiple input parameters by enclosing them in parentheses. The method returns all the elements in the `numbers` array until it finds a number whose value is less than its ordinal position in the array:

C#

```
int[] numbers = { 5, 4, 1, 3, 9, 8, 6, 7, 2, 0 };
var firstSmallNumbers = numbers.TakeWhile((n, index) => n >= index);
Console.WriteLine(string.Join(" ", firstSmallNumbers));
// Output:
// 5 4
```

You don't use lambda expressions directly in query expressions, but you can use them in method calls within query expressions, as the following example shows:

C#

```
var numberSets = new List<int[]>
{
    new[] { 1, 2, 3, 4, 5 },
    new[] { 0, 0, 0 },
    new[] { 9, 8 },
    new[] { 1, 0, 1, 0, 1, 0, 1, 0 }
};

var setsWithManyPositives =
    from numberSet in numberSets
    where numberSet.Count(n => n > 0) > 3
    select numberSet;

foreach (var numberSet in setsWithManyPositives)
{
```

```
    Console.WriteLine(string.Join(" ", numberSet));
}
// Output:
// 1 2 3 4 5
// 1 0 1 0 1 0 1 0
```

Type inference in lambda expressions

When writing lambdas, you often don't have to specify a type for the input parameters because the compiler can infer the type based on the lambda body, the parameter types, and other factors as described in the C# language specification. For most of the standard query operators, the first input is the type of the elements in the source sequence. If you're querying an `IEnumerable<Customer>`, then the input variable is inferred to be a `Customer` object, which means you have access to its methods and properties:

C#

```
customers.Where(c => c.City == "London");
```

The general rules for type inference for lambdas are as follows:

- The lambda must contain the same number of parameters as the delegate type.
- Each input parameter in the lambda must be implicitly convertible to its corresponding delegate parameter.
- The return value of the lambda (if any) must be implicitly convertible to the delegate's return type.

Natural type of a lambda expression

A lambda expression in itself doesn't have a type because the common type system has no intrinsic concept of "lambda expression." However, it's sometimes convenient to speak informally of the "type" of a lambda expression. That informal "type" refers to the delegate type or Expression type to which the lambda expression is converted.

A lambda expression can have a *natural type*. Instead of forcing you to declare a delegate type, such as `Func<...>` or `Action<...>` for a lambda expression, the compiler can infer the delegate type from the lambda expression. For example, consider the following declaration:

C#

```
var parse = (string s) => int.Parse(s);
```

The compiler can infer `parse` to be a `Func<string, int>`. The compiler chooses an available `Func` or `Action` delegate, if a suitable one exists. Otherwise, it synthesizes a delegate type. For example, the delegate type is synthesized if the lambda expression has `ref` parameters. When a lambda expression has a natural type, it can be assigned to a less explicit type, such as System.Object or System.Delegate:

C#

```
object parse = (string s) => int.Parse(s);   // Func<string, int>
Delegate parse = (string s) => int.Parse(s); // Func<string, int>
```

Method groups (that is, method names without parameter lists) with exactly one overload have a natural type:

C#

```
var read = Console.Read; // Just one overload; Func<int> inferred
var write = Console.Write; // ERROR: Multiple overloads, can't choose
```

If you assign a lambda expression to System.Linq.Expressions.LambdaExpression, or System.Linq.Expressions.Expression, and the lambda has a natural delegate type, the expression has a natural type of System.Linq.Expressions.Expression<TDelegate>, with the natural delegate type used as the argument for the type parameter:

C#

```
LambdaExpression parseExpr = (string s) => int.Parse(s); //
Expression<Func<string, int>>
Expression parseExpr = (string s) => int.Parse(s);       //
Expression<Func<string, int>>
```

Not all lambda expressions have a natural type. Consider the following declaration:

C#

```
var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda
```

The compiler can't infer a parameter type for `s`. When the compiler can't infer a natural type, you must declare the type:

C#

```
Func<string, int> parse = s => int.Parse(s);
```

Explicit return type

Typically, the return type of a lambda expression is obvious and inferred. For some expressions that doesn't work:

C#

```
var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type
```

You can specify the return type of a lambda expression before the input parameters. When you specify an explicit return type, you must parenthesize the input parameters:

C#

```
var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>
```

Attributes

You can add attributes to a lambda expression and its parameters. The following example shows how to add attributes to a lambda expression:

C#

```
Func<string?, int?> parse = [ProvidesNullCheck] (s) => (s is not null) ?
int.Parse(s) : null;
```

You can also add attributes to the input parameters or return value, as the following example shows:

C#

```
var concat = ([DisallowNull] string a, [DisallowNull] string b) => a + b;
var inc = [return: NotNullIfNotNull(nameof(s))] (int? s) => s.HasValue ? s++
: null;
```

As the preceding examples show, you must parenthesize the input parameters when you add attributes to a lambda expression or its parameters.

> **Important**
>
> Lambda expressions are invoked through the underlying delegate type. That invocation is different than methods and local functions. The delegate's `Invoke` method doesn't check attributes on the lambda expression. Attributes don't have any effect when the lambda expression is invoked. Attributes on lambda expressions are useful for code analysis, and can be discovered via reflection. One consequence of this decision is that the System.Diagnostics.ConditionalAttribute can't be applied to a lambda expression.

Capture of outer variables and variable scope in lambda expressions

Lambdas can refer to *outer variables*. These *outer variables* are the variables that are in scope in the method that defines the lambda expression, or in scope in the type that contains the lambda expression. Variables that are captured in this manner are stored for use in the lambda expression even if the variables would otherwise go out of scope and be garbage collected. An outer variable must be definitely assigned before it can be consumed in a lambda expression. The following example demonstrates these rules:

C#

```
public static class VariableScopeWithLambdas
{
    public class VariableCaptureGame
    {
        internal Action<int>? updateCapturedLocalVariable;
        internal Func<int, bool>? isEqualToCapturedLocalVariable;

        public void Run(int input)
        {
            int j = 0;

            updateCapturedLocalVariable = x =>
            {
                j = x;
                bool result = j > input;
                Console.WriteLine($"{j} is greater than {input}: {result}");
            };

            isEqualToCapturedLocalVariable = x => x == j;

            Console.WriteLine($"Local variable before lambda invocation:
{j}");
            updateCapturedLocalVariable(10);
```

```
            Console.WriteLine($"Local variable after lambda invocation:
{j}");
        }
    }

    public static void Main()
    {
        var game = new VariableCaptureGame();

        int gameInput = 5;
        game.Run(gameInput);

        int jTry = 10;
        bool result = game.isEqualToCapturedLocalVariable!(jTry);
        Console.WriteLine($"Captured local variable is equal to {jTry}:
{result}");

        int anotherJ = 3;
        game.updateCapturedLocalVariable!(anotherJ);

        bool equalToAnother = game.isEqualToCapturedLocalVariable(anotherJ);
        Console.WriteLine($"Another lambda observes a new value of captured
variable: {equalToAnother}");
    }
    // Output:
    // Local variable before lambda invocation: 0
    // 10 is greater than 5: True
    // Local variable after lambda invocation: 10
    // Captured local variable is equal to 10: True
    // 3 is greater than 5: False
    // Another lambda observes a new value of captured variable: True
}
```

The following rules apply to variable scope in lambda expressions:

- A variable that is captured isn't garbage-collected until the delegate that references it becomes eligible for garbage collection.
- Variables introduced within a lambda expression aren't visible in the enclosing method.
- A lambda expression can't directly capture an in, ref, or out parameter from the enclosing method.
- A return statement in a lambda expression doesn't cause the enclosing method to return.
- A lambda expression can't contain a goto, break, or continue statement if the target of that jump statement is outside the lambda expression block. It's also an error to have a jump statement outside the lambda expression block if the target is inside the block.

You can apply the `static` modifier to a lambda expression to prevent unintentional capture of local variables or instance state by the lambda:

```
C#
```

```
Func<double, double> square = static x => x * x;
```

A static lambda can't capture local variables or instance state from enclosing scopes, but can reference static members and constant definitions.

C# language specification

For more information, see the Anonymous function expressions section of the C# language specification.

For more information about these features, see the following feature proposal notes:

- Lambda discard parameters
- Static anonymous functions
- Lambda improvements

See also

- Use local function instead of lambda (style rule IDE0039)
- C# operators and expressions
- LINQ (Language-Integrated Query)
- Expression trees
- Local functions vs. lambda expressions
- LINQ sample queries
- XQuery sample

Pattern matching - the `is` and `switch` expressions, and operators `and`, `or`, and `not` in patterns

Article • 02/18/2025

You use the is expression, the switch statement, and the switch expression to match an input expression against any number of characteristics. C# supports multiple patterns, including declaration, type, constant, relational, property, list, var, and discard. Patterns can be combined using Boolean logic keywords `and`, `or`, and `not`.

The following C# expressions and statements support pattern matching:

- is expression
- switch statement
- switch expression

In those constructs, you can match an input expression against any of the following patterns:

- Declaration pattern: check the run-time type of an expression and, if a match succeeds, assign an expression result to a declared variable.
- Type pattern: check the run-time type of an expression.
- Constant pattern: test that an expression result equals a specified constant.
- Relational patterns: compare an expression result with a specified constant.
- Logical patterns: test that an expression matches a logical combination of patterns.
- Property pattern: test that an expression's properties or fields match nested patterns.
- Positional pattern: deconstruct an expression result and test if the resulting values match nested patterns.
- var pattern: match any expression and assign its result to a declared variable.
- Discard pattern: match any expression.
- List patterns: test that a sequence of elements matches corresponding nested patterns. Introduced in C# 11.

Logical, property, positional, and list patterns are *recursive* patterns. That is, they can contain *nested* patterns.

For the example of how to use those patterns to build a data-driven algorithm, see Tutorial: Use pattern matching to build type-driven and data-driven algorithms.

Declaration and type patterns

You use declaration and type patterns to check if the run-time type of an expression is compatible with a given type. With a declaration pattern, you can also declare a new local variable. When a declaration pattern matches an expression, that variable is assigned a converted expression result, as the following example shows:

```
C#

object greeting = "Hello, World!";
if (greeting is string message)
{
    Console.WriteLine(message.ToLower());  // output: hello, world!
}
```

A *declaration pattern* with type `T` matches an expression when an expression result is non-null and any of the following conditions are true:

- The run-time type of an expression result is `T`.
- The type `T` is a `ref struct` type and there is an identity conversion from the expression to `T`.
- The run-time type of an expression result derives from type `T`, implements interface `T`, or another implicit reference conversion exists from it to `T`. The following example demonstrates two cases when this condition is true:

  ```
  C#

  var numbers = new int[] { 10, 20, 30 };
  Console.WriteLine(GetSourceLabel(numbers));  // output: 1

  var letters = new List<char> { 'a', 'b', 'c', 'd' };
  Console.WriteLine(GetSourceLabel(letters));  // output: 2

  static int GetSourceLabel<T>(IEnumerable<T> source) => source switch
  {
      Array array => 1,
      ICollection<T> collection => 2,
      _ => 3,
  };
  ```

  In the preceding example, at the first call to the `GetSourceLabel` method, the first pattern matches an argument value because the argument's run-time type `int[]` derives from the Array type. At the second call to the `GetSourceLabel` method, the argument's run-time type List<T> doesn't derive from the Array type but implements the ICollection<T> interface.

- The run-time type of an expression result is a nullable value type with the underlying type `T`.
- A boxing or unboxing conversion exists from the run-time type of an expression result to type `T`.

The following example demonstrates the last two conditions:

```
C#

int? xNullable = 7;
int y = 23;
object yBoxed = y;
if (xNullable is int a && yBoxed is int b)
{
    Console.WriteLine(a + b);  // output: 30
}
```

If you want to check only the type of an expression, you can use a discard `_` in place of a variable's name, as the following example shows:

```
C#

public abstract class Vehicle {}
public class Car : Vehicle {}
public class Truck : Vehicle {}

public static class TollCalculator
{
    public static decimal CalculateToll(this Vehicle vehicle) => vehicle
switch
    {
        Car _ => 2.00m,
        Truck _ => 7.50m,
        null => throw new ArgumentNullException(nameof(vehicle)),
        _ => throw new ArgumentException("Unknown type of a vehicle",
nameof(vehicle)),
    };
}
```

For that purpose you can use a *type pattern*, as the following example shows:

```
C#

public static decimal CalculateToll(this Vehicle vehicle) => vehicle switch
{
    Car => 2.00m,
    Truck => 7.50m,
    null => throw new ArgumentNullException(nameof(vehicle)),
    _ => throw new ArgumentException("Unknown type of a vehicle",
```

```
nameof(vehicle)),
};
```

Like a declaration pattern, a type pattern matches an expression when an expression result is non-null and its run-time type satisfies any of the preceding conditions.

To check for non-null, you can use a negated `null` constant pattern, as the following example shows:

C#

```
if (input is not null)
{
    // ...
}
```

For more information, see the Declaration pattern and Type pattern sections of the feature proposal notes.

Constant pattern

The *constant pattern* is an alternative syntax for `==` when the right operand is a constant. You use a *constant pattern* to test if an expression result equals a specified constant, as the following example shows:

C#

```
public static decimal GetGroupTicketPrice(int visitorCount) => visitorCount
switch
{
    1 => 12.0m,
    2 => 20.0m,
    3 => 27.0m,
    4 => 32.0m,
    0 => 0.0m,
    _ => throw new ArgumentException($"Not supported number of visitors:
{visitorCount}", nameof(visitorCount)),
};
```

In a constant pattern, you can use any constant expression, such as:

- an integer or floating-point numerical literal
- a char
- a string literal.
- a Boolean value `true` or `false`

- an enum value
- the name of a declared const field or local
- `null`

The expression must be a type that is convertible to the constant type, with one exception: An expression whose type is `Span<char>` or `ReadOnlySpan<char>` can be matched against constant strings in C# 11 and later versions.

Use a constant pattern to check for `null`, as the following example shows:

```
C#

if (input is null)
{
    return;
}
```

The compiler guarantees that no user-overloaded equality operator `==` is invoked when expression `x is null` is evaluated.

You can use a negated `null` constant pattern to check for non-null, as the following example shows:

```
C#

if (input is not null)
{
    // ...
}
```

For more information, see the Constant pattern section of the feature proposal note.

Relational patterns

You use a *relational pattern* to compare an expression result with a constant, as the following example shows:

```
C#

Console.WriteLine(Classify(13));  // output: Too high
Console.WriteLine(Classify(double.NaN));  // output: Unknown
Console.WriteLine(Classify(2.4));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
```

```
    < -4.0 => "Too low",
    > 10.0 => "Too high",
    double.NaN => "Unknown",
    _ => "Acceptable",
};
```

In a relational pattern, you can use any of the relational operators `<`, `>`, `<=`, or `>=`. The right-hand part of a relational pattern must be a constant expression. The constant expression can be of an integer, floating-point, char, or enum type.

To check if an expression result is in a certain range, match it against a conjunctive and pattern, as the following example shows:

C#

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 3, 14)));  // output:
spring
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 7, 19)));  // output:
summer
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 2, 17)));  // output:
winter

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    >= 3 and < 6 => "spring",
    >= 6 and < 9 => "summer",
    >= 9 and < 12 => "autumn",
    12 or (>= 1 and < 3) => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date with
unexpected month: {date.Month}."),
};
```

If an expression result is `null` or fails to convert to the type of a constant by a nullable or unboxing conversion, a relational pattern doesn't match an expression.

For more information, see the Relational patterns section of the feature proposal note.

Logical patterns

You use the `not`, `and`, and `or` pattern combinators to create the following *logical patterns*:

- *Negation* `not` pattern that matches an expression when the negated pattern doesn't match the expression. The following example shows how you can negate a constant `null` pattern to check if an expression is non-null:

  ```
  ```

C#

```
if (input is not null)
{
    // ...
}
```

- *Conjunctive* `and` pattern that matches an expression when both patterns match the expression. The following example shows how you can combine relational patterns to check if a value is in a certain range:

C#

```
Console.WriteLine(Classify(13));  // output: High
Console.WriteLine(Classify(-100));  // output: Too low
Console.WriteLine(Classify(5.7));  // output: Acceptable

static string Classify(double measurement) => measurement switch
{
    < -40.0 => "Too low",
    >= -40.0 and < 0 => "Low",
    >= 0 and < 10.0 => "Acceptable",
    >= 10.0 and < 20.0 => "High",
    >= 20.0 => "Too high",
    double.NaN => "Unknown",
};
```

- *Disjunctive* `or` pattern that matches an expression when either pattern matches the expression, as the following example shows:

C#

```
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 1, 19)));  //
output: winter
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 10, 9)));  //
output: autumn
Console.WriteLine(GetCalendarSeason(new DateTime(2021, 5, 11)));  //
output: spring

static string GetCalendarSeason(DateTime date) => date.Month switch
{
    3 or 4 or 5 => "spring",
    6 or 7 or 8 => "summer",
    9 or 10 or 11 => "autumn",
    12 or 1 or 2 => "winter",
    _ => throw new ArgumentOutOfRangeException(nameof(date), $"Date
with unexpected month: {date.Month}."),
};
```

As the preceding example shows, you can repeatedly use the pattern combinators in a pattern.

Precedence and order of checking

The pattern combinators are ordered based on the binding order of expressions as follows:

- `not`
- `and`
- `or`

The `not` pattern binds to its operand first. The `and` pattern binds after any `not` pattern expression binding. The `or` pattern binds after all `not` and `and` patterns are bound to operands. The following example tries to match all characters that aren't lower case letters, `a` - `z`. It has an error, because the `not` pattern binds before the `and` pattern:

C#

```
// Incorrect pattern. `not` binds before `and`
static bool IsNotLowerCaseLetter(char c) => c is not >= 'a' and <= 'z';
```

The default binding means the previous example is parsed as the following example:

C#

```
// The default binding without parentheses is shows in this method. `not`
binds before `and`
static bool IsNotLowerCaseLetterDefaultBinding(char c) => c is ((not >= 'a')
and <= 'z');
```

To fix it, you must specify that you want the `not` to bind to the `>= 'a' and <= 'z'` expression:

C#

```
// Correct pattern. Force `and` before `not`
static bool IsNotLowerCaseLetterParentheses(char c) => c is not (>= 'a' and
<= 'z');
```

Adding parentheses becomes more important as your patterns become more complicated. In general, use parentheses to clarify your patterns for other developers, as the following example shows:

C#

```
static bool IsLetter(char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <=
'Z');
```

> **Note**
>
> The order in which patterns having the same binding order are checked is undefined. At run time, the right-hand nested patterns of multiple `or` patterns and multiple `and` patterns can be checked first.

For more information, see the Pattern combinators section of the feature proposal note.

Property pattern

You use a *property pattern* to match an expression's properties or fields against nested patterns, as the following example shows:

C#

```
static bool IsConferenceDay(DateTime date) => date is { Year: 2020, Month:
5, Day: 19 or 20 or 21 };
```

A property pattern matches an expression when an expression result is non-null and every nested pattern matches the corresponding property or field of the expression result.

You can also add a run-time type check and a variable declaration to a property pattern, as the following example shows:

C#

```
Console.WriteLine(TakeFive("Hello, world!"));  // output: Hello
Console.WriteLine(TakeFive("Hi!"));  // output: Hi!
Console.WriteLine(TakeFive(new[] { '1', '2', '3', '4', '5', '6', '7' }));
// output: 12345
Console.WriteLine(TakeFive(new[] { 'a', 'b', 'c' }));  // output: abc

static string TakeFive(object input) => input switch
{
    string { Length: >= 5 } s => s.Substring(0, 5),
    string s => s,

    ICollection<char> { Count: >= 5 } symbols => new
string(symbols.Take(5).ToArray()),
```

```
    ICollection<char> symbols => new string(symbols.ToArray()),

    null => throw new ArgumentNullException(nameof(input)),
    _ => throw new ArgumentException("Not supported input type."),
};
```

A property pattern is a recursive pattern. You can use any pattern as a nested pattern. Use a property pattern to match parts of data against nested patterns, as the following example shows:

C#

```
public record Point(int X, int Y);
public record Segment(Point Start, Point End);

static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start: { Y: 0 } } or { End: { Y: 0 } };
```

The preceding example uses the `or` pattern combinator and record types.

You can reference nested properties or fields within a property pattern. This capability is known as an *extended property pattern*. For example, you can refactor the method from the preceding example into the following equivalent code:

C#

```
static bool IsAnyEndOnXAxis(Segment segment) =>
    segment is { Start.Y: 0 } or { End.Y: 0 };
```

For more information, see the Property pattern section of the feature proposal note and the Extended property patterns feature proposal note.

> **Tip**
>
> You can use the **Simplify property pattern (IDE0170)** style rule to improve code readability by suggesting places to use extended property patterns.

Positional pattern

You use a *positional pattern* to deconstruct an expression result and match the resulting values against the corresponding nested patterns, as the following example shows:

C#

```
public readonly struct 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);
}

static string Classify(Point point) => point switch
{
    (0, 0) => "Origin",
    (1, 0) => "positive X basis end",
    (0, 1) => "positive Y basis end",
    _ => "Just a point",
};
```

At the preceding example, the type of an expression contains the Deconstruct method, which is used to deconstruct an expression result.

> **ⓘ Important**
>
> The order of members in a positional pattern must match the order of parameters in the `Deconstruct` method. That's because the code generated for the positional pattern calls the `Deconstruct` method.

You can also match expressions of tuple types against positional patterns. In that way, you can match multiple inputs against various patterns, as the following example shows:

C#

```
static decimal GetGroupTicketPriceDiscount(int groupSize, DateTime
visitDate)
    => (groupSize, visitDate.DayOfWeek) switch
    {
        (<= 0, _) => throw new ArgumentException("Group size must be
positive."),
        (_, DayOfWeek.Saturday or DayOfWeek.Sunday) => 0.0m,
        (>= 5 and < 10, DayOfWeek.Monday) => 20.0m,
        (>= 10, DayOfWeek.Monday) => 30.0m,
        (>= 5 and < 10, _) => 12.0m,
        (>= 10, _) => 15.0m,
        _ => 0.0m,
    };
```

The preceding example uses relational and logical patterns.

You can use the names of tuple elements and `Deconstruct` parameters in a positional pattern, as the following example shows:

```
C#
```

```
var numbers = new List<int> { 1, 2, 3 };
if (SumAndCount(numbers) is (Sum: var sum, Count: > 0))
{
    Console.WriteLine($"Sum of [{string.Join(" ", numbers)}] is {sum}");  //
output: Sum of [1 2 3] is 6
}

static (double Sum, int Count) SumAndCount(IEnumerable<int> numbers)
{
    int sum = 0;
    int count = 0;
    foreach (int number in numbers)
    {
        sum += number;
        count++;
    }
    return (sum, count);
}
```

You can also extend a positional pattern in any of the following ways:

- Add a run-time type check and a variable declaration, as the following example shows:

  ```
  C#
  ```

  ```
  public record Point2D(int X, int Y);
  public record Point3D(int X, int Y, int Z);

  static string PrintIfAllCoordinatesArePositive(object point) => point
  switch
  {
      Point2D (> 0, > 0) p => p.ToString(),
      Point3D (> 0, > 0, > 0) p => p.ToString(),
      _ => string.Empty,
  };
  ```

  The preceding example uses positional records that implicitly provide the `Deconstruct` method.

- Use a property pattern within a positional pattern, as the following example shows:

  ```
  C#
  ```

```
public record WeightedPoint(int X, int Y)
{
    public double Weight { get; set; }
}

static bool IsInDomain(WeightedPoint point) => point is (>= 0, >= 0) {
Weight: >= 0.0 };
```

- Combine two preceding usages, as the following example shows:

```
C#

if (input is WeightedPoint (> 0, > 0) { Weight: > 0.0 } p)
{
    // ..
}
```

A positional pattern is a recursive pattern. That is, you can use any pattern as a nested pattern.

For more information, see the Positional pattern section of the feature proposal note.

`var` pattern

You use a *`var` pattern* to match any expression, including `null`, and assign its result to a new local variable, as the following example shows:

```
C#

static bool IsAcceptable(int id, int absLimit) =>
    SimulateDataFetch(id) is var results
    && results.Min() >= -absLimit
    && results.Max() <= absLimit;

static int[] SimulateDataFetch(int id)
{
    var rand = new Random();
    return Enumerable
               .Range(start: 0, count: 5)
               .Select(s => rand.Next(minValue: -10, maxValue: 11))
               .ToArray();
}
```

A `var` pattern is useful when you need a temporary variable within a Boolean expression to hold the result of intermediate calculations. You can also use a `var` pattern when you

need to perform more checks in `when` case guards of a `switch` expression or statement, as the following example shows:

C#

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

static Point Transform(Point point) => point switch
{
    var (x, y) when x < y => new Point(-x, y),
    var (x, y) when x > y => new Point(x, -y),
    var (x, y) => new Point(x, y),
};

static void TestTransform()
{
    Console.WriteLine(Transform(new Point(1, 2)));  // output: Point { X =
-1, Y = 2 }
    Console.WriteLine(Transform(new Point(5, 2)));  // output: Point { X =
5, Y = -2 }
}
```

In the preceding example, pattern `var (x, y)` is equivalent to a positional pattern `(var x, var y)`.

In a `var` pattern, the type of a declared variable is the compile-time type of the expression that is matched against the pattern.

For more information, see the Var pattern section of the feature proposal note.

Discard pattern

You use a *discard pattern* `_` to match any expression, including `null`, as the following example shows:

C#

```
Console.WriteLine(GetDiscountInPercent(DayOfWeek.Friday));  // output: 5.0
Console.WriteLine(GetDiscountInPercent(null));  // output: 0.0
Console.WriteLine(GetDiscountInPercent((DayOfWeek)10));  // output: 0.0

static decimal GetDiscountInPercent(DayOfWeek? dayOfWeek) => dayOfWeek
switch
{
    DayOfWeek.Monday => 0.5m,
    DayOfWeek.Tuesday => 12.5m,
    DayOfWeek.Wednesday => 7.5m,
    DayOfWeek.Thursday => 12.5m,
```

```
    DayOfWeek.Friday => 5.0m,
    DayOfWeek.Saturday => 2.5m,
    DayOfWeek.Sunday => 2.0m,
    _ => 0.0m,
};
```

In the preceding example, a discard pattern is used to handle `null` and any integer value that doesn't have the corresponding member of the DayOfWeek enumeration. That guarantees that a `switch` expression in the example handles all possible input values. If you don't use a discard pattern in a `switch` expression and none of the expression's patterns matches an input, the runtime throws an exception. The compiler generates a warning if a `switch` expression doesn't handle all possible input values.

A discard pattern can't be a pattern in an `is` expression or a `switch` statement. In those cases, to match any expression, use a var pattern with a discard: `var _`. A discard pattern can be a pattern in a `switch` expression.

For more information, see the Discard pattern section of the feature proposal note.

Parenthesized pattern

You can put parentheses around any pattern. Typically, you do that to emphasize or change the precedence in logical patterns, as the following example shows:

C#

```
if (input is not (float or double))
{
    return;
}
```

List patterns

Beginning with C# 11, you can match an array or a list against a *sequence* of patterns, as the following example shows:

C#

```
int[] numbers = { 1, 2, 3 };

Console.WriteLine(numbers is [1, 2, 3]);  // True
Console.WriteLine(numbers is [1, 2, 4]);  // False
```

```
Console.WriteLine(numbers is [1, 2, 3, 4]);  // False
Console.WriteLine(numbers is [0 or 1, <= 2, >= 3]);  // True
```

As the preceding example shows, a list pattern is matched when each nested pattern matches the corresponding element of an input sequence. You can use any pattern within a list pattern. To match any element, use the discard pattern or, if you also want to capture the element, the var pattern, as the following example shows:

C#

```
List<int> numbers = new() { 1, 2, 3 };

if (numbers is [var first, _, _])
{
    Console.WriteLine($"The first element of a three-item list is
{first}.");
}
// Output:
// The first element of a three-item list is 1.
```

The preceding examples match a whole input sequence against a list pattern. To match elements only at the start or/and the end of an input sequence, use the *slice pattern* `..`, as the following example shows:

C#

```
Console.WriteLine(new[] { 1, 2, 3, 4, 5 } is [> 0, > 0, ..]);  // True
Console.WriteLine(new[] { 1, 1 } is [_, _, ..]);  // True
Console.WriteLine(new[] { 0, 1, 2, 3, 4 } is [> 0, > 0, ..]);  // False
Console.WriteLine(new[] { 1 } is [1, 2, ..]);  // False

Console.WriteLine(new[] { 1, 2, 3, 4 } is [.., > 0, > 0]);  // True
Console.WriteLine(new[] { 2, 4 } is [.., > 0, 2, 4]);  // False
Console.WriteLine(new[] { 2, 4 } is [.., 2, 4]);  // True

Console.WriteLine(new[] { 1, 2, 3, 4 } is [>= 0, .., 2 or 4]);  // True
Console.WriteLine(new[] { 1, 0, 0, 1 } is [1, 0, .., 0, 1]);  // True
Console.WriteLine(new[] { 1, 0, 1 } is [1, 0, .., 0, 1]);  // False
```

A slice pattern matches zero or more elements. You can use at most one slice pattern in a list pattern. The slice pattern can only appear in a list pattern.

You can also nest a subpattern within a slice pattern, as the following example shows:

C#

```
void MatchMessage(string message)
{
```

```
    var result = message is ['a' or 'A', .. var s, 'a' or 'A']
        ? $"Message {message} matches; inner part is {s}."
        : $"Message {message} doesn't match.";
    Console.WriteLine(result);
}

MatchMessage("aBBA");  // output: Message aBBA matches; inner part is BB.
MatchMessage("apron");  // output: Message apron doesn't match.

void Validate(int[] numbers)
{
    var result = numbers is [< 0, .. { Length: 2 or 4 }, > 0] ? "valid" :
"not valid";
    Console.WriteLine(result);
}

Validate(new[] { -1, 0, 1 });  // output: not valid
Validate(new[] { -1, 0, 0, 1 });  // output: valid
```

For more information, see the List patterns feature proposal note.

C# language specification

For more information, see the Patterns and pattern matching section of the C# language specification.

For information about features added in C# 8 and later, see the following feature proposal notes:

- Recursive pattern matching
- Pattern-matching updates
- Extended property patterns
- C# 11 - List patterns
- C# 11 - Pattern match Span<char> on string literal

See also

- C# operators and expressions
- Pattern matching overview
- Tutorial: Use pattern matching to build type-driven and data-driven algorithms

Addition operators - `+` and `+=`

06/13/2025

The built-in integral and floating-point numeric types, the string type, and delegate types all support the `+` and `+=` operators.

For information about the arithmetic `+` operator, see the Unary plus and minus operators and Addition operator + sections of the Arithmetic operators article.

String concatenation

When one or both operands are of type string, the `+` operator concatenates the string representations of its operands (the string representation of `null` is an empty string):

```
C#
```

```
Console.WriteLine("Forgot" + "white space");
Console.WriteLine("Probably the oldest constant: " + Math.PI);
Console.WriteLine(null + "Nothing to add.");
// Output:
// Forgotwhite space
// Probably the oldest constant: 3.14159265358979
// Nothing to add.
```

String interpolation provides a more convenient way to format strings:

```
C#
```

```
Console.WriteLine($"Probably the oldest constant: {Math.PI:F2}");
// Output:
// Probably the oldest constant: 3.14
```

You can use string interpolation to initialize a constant string when all the expressions used for placeholders are also constant strings.

Beginning with C# 11, the `+` operator performs string concatenation for UTF-8 literal strings. This operator concatenates two `ReadOnlySpan<byte>` objects.

Delegate combination

For operands of the same delegate type, the `+` operator returns a new delegate instance that, when invoked, invokes the left-hand operand and then invokes the right-hand operand. If any

of the operands is `null`, the `+` operator returns the value of another operand (which also might be `null`). The following example shows how delegates can be combined with the `+` operator:

```
C#

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
Action ab = a + b;
ab();  // output: ab
```

To perform delegate removal, use the - operator.

For more information about delegate types, see Delegates.

Addition assignment operator +=

An expression using the `+=` operator, such as

```
C#

x += y
```

Is equivalent to:

```
C#

x = x + y
```

Except that `x` is only evaluated once.

The following example demonstrates the usage of the `+=` operator:

```
C#

int i = 5;
i += 9;
Console.WriteLine(i);
// Output: 14

string story = "Start. ";
story += "End.";
Console.WriteLine(story);
// Output: Start. End.

Action printer = () => Console.Write("a");
```

```
printer();  // output: a

Console.WriteLine();
printer += () => Console.Write("b");
printer();  // output: ab
```

You also use the `+=` operator to specify an event handler method when you subscribe to an event. For more information, see How to: subscribe to and unsubscribe from events.

Operator overloadability

A user-defined type can overload the `+` operator. When a binary `+` operator is overloaded, the `+=` operator is also implicitly overloaded. Beginning with C# 14, a user-defined type can explicitly overload the `+=` operator to provide a more efficient implementation. Typically, a type overloads the `+=` operator because the value can be updated in place, rather than allocate a new instance to hold the result of the addition. If a type doesn't provide an explicit overload, the compiler generates the implicit overload.

C# language specification

For more information, see the Unary plus operator and Addition operator sections of the C# language specification. For more information on overloading the compound assignment operators in C# 14 and later, see the user defined compound assignment feature specification.

See also

- C# operators and expressions
- How to concatenate multiple strings
- Events
- Arithmetic operators
- - and -= operators

- and -= operators - subtraction (minus)

06/13/2025

The built-in integral and floating-point numeric types and delegate types all support the `-` and `-=` operators.

For information about the arithmetic `-` operator, see the Unary plus and minus operators and Subtraction operator - sections of the Arithmetic operators article.

Delegate removal

For operands of the same delegate type, the `-` operator returns a delegate instance that is calculated as follows:

- If both operands are non-null and the invocation list of the right-hand operand is a proper contiguous sublist of the invocation list of the left-hand operand, the result of the operation is a new invocation list that is obtained by removing the right-hand operand's entries from the invocation list of the left-hand operand. If the right-hand operand's list matches multiple contiguous sublists in the left-hand operand's list, only the right-most matching sublist is removed. If removal results in an empty list, the result is `null`.

  C#

  ```
  Action a = () => Console.Write("a");
  Action b = () => Console.Write("b");

  var abbaab = a + b + b + a + a + b;
  abbaab();  // output: abbaab
  Console.WriteLine();

  var ab = a + b;
  var abba = abbaab - ab;
  abba();  // output: abba
  Console.WriteLine();

  var nihil = abbaab - abbaab;
  Console.WriteLine(nihil is null);  // output: True
  ```

- If the invocation list of the right-hand operand isn't a proper contiguous sublist of the invocation list of the left-hand operand, the result of the operation is the left-hand operand. For example, removing a delegate that isn't part of the multicast delegate does nothing and results in the unchanged multicast delegate.

  C#

```
Action a = () => Console.Write("a");
Action b = () => Console.Write("b");

var abbaab = a + b + b + a + a + b;
var aba = a + b + a;

var first = abbaab - aba;
first();  // output: abbaab
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(abbaab, first));  // output: True

Action a2 = () => Console.Write("a");
var changed = aba - a;
changed();  // output: ab
Console.WriteLine();
var unchanged = aba - a2;
unchanged();  // output: aba
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(aba, unchanged));  // output: True
```

The preceding example also demonstrates that during delegate removal delegate instances are compared. For example, delegates that are produced from evaluation of identical lambda expressions aren't equal. For more information about delegate equality, see the Delegate equality operators section of the C# language specification.

- If the left-hand operand is `null`, the result of the operation is `null`. If the right-hand operand is `null`, the result of the operation is the left-hand operand.

C#

```
Action a = () => Console.Write("a");

var nothing = null - a;
Console.WriteLine(nothing is null);  // output: True

var first = a - null;
a();  // output: a
Console.WriteLine();
Console.WriteLine(object.ReferenceEquals(first, a));  // output: True
```

To combine delegates, use the + operator.

For more information about delegate types, see Delegates.

Subtraction assignment operator -=

An expression using the `-=` operator, such as

```
C#

x -= y
```

Is equivalent to

```
C#

x = x - y
```

Except that `x` is only evaluated once.

The following example demonstrates the usage of the `--` operator:

```
C#

int i = 5;
i -= 9;
Console.WriteLine(i);
// Output: -4

Action a = () => Console.Write("a");
Action b = () => Console.Write("b");
var printer = a + b + a;
printer();  // output: aba

Console.WriteLine();
printer -= a;
printer();  // output: ab
```

You also use the `--` operator to specify an event handler method to remove when you unsubscribe from an event. For more information, see How to subscribe to and unsubscribe from events.

Operator overloadability

A user-defined type can overload the `-` operator. When a binary `-` operator is overloaded, the `--` operator is also implicitly overloaded. Beginning with C# 14, a user-defined type can explicitly overload the `--` operator to provide a more efficient implementation. Typically, a type overloads the `--` operator because the value can be updated in place, rather than allocating a new instance to store the result of the subtraction. If a type doesn't provide an explicit overload, the compiler generates the implicit overload.

C# language specification

For more information, see the Unary minus operator and Subtraction operator sections of the C# language specification. For more information on overloading the compound assignment operators in C# 14 and later, see the user defined compound assignment feature specification.

See also

- C# operators and expressions
- Events
- Arithmetic operators
- + and += operators

?: operator - the ternary conditional operator

Article • 07/25/2023

The conditional operator `?:`, also known as the ternary conditional operator, evaluates a Boolean expression and returns the result of one of the two expressions, depending on whether the Boolean expression evaluates to `true` or `false`, as the following example shows:

C#

```
string GetWeatherDisplay(double tempInCelsius) => tempInCelsius < 20.0 ?
"Cold." : "Perfect!";

Console.WriteLine(GetWeatherDisplay(15));  // output: Cold.
Console.WriteLine(GetWeatherDisplay(27));  // output: Perfect!
```

As the preceding example shows, the syntax for the conditional operator is as follows:

C#

```
condition ? consequent : alternative
```

The `condition` expression must evaluate to `true` or `false`. If `condition` evaluates to `true`, the `consequent` expression is evaluated, and its result becomes the result of the operation. If `condition` evaluates to `false`, the `alternative` expression is evaluated, and its result becomes the result of the operation. Only `consequent` or `alternative` is evaluated. Conditional expressions are target-typed. That is, if a target type of a conditional expression is known, the types of `consequent` and `alternative` must be implicitly convertible to the target type, as the following example shows:

C#

```
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

int? x = condition ? 12 : null;

IEnumerable<int> xs = x is null ? new List<int>() { 0, 1 } : new int[] { 2,
3 };
```

If a target type of a conditional expression is unknown (for example, when you use the var keyword) or the type of `consequent` and `alternative` must be the same or there must be an implicit conversion from one type to the other:

```C#
var rand = new Random();
var condition = rand.NextDouble() > 0.5;

var x = condition ? 12 : (int?)null;
```

The conditional operator is right-associative, that is, an expression of the form

```C#
a ? b : c ? d : e
```

is evaluated as

```C#
a ? b : (c ? d : e)
```

> **Tip**
>
> You can use the following mnemonic device to remember how the conditional operator is evaluated:
>
> ```text
> is this condition true ? yes : no
> ```

Conditional ref expression

A conditional ref expression conditionally returns a variable reference, as the following example shows:

```C#
int[] smallArray = {1, 2, 3, 4, 5};
int[] largeArray = {10, 20, 30, 40, 50};

int index = 7;
```

```
ref int refValue = ref ((index < 5) ? ref smallArray[index] : ref
largeArray[index - 5]);
refValue = 0;

index = 2;
((index < 5) ? ref smallArray[index] : ref largeArray[index - 5]) = 100;

Console.WriteLine(string.Join(" ", smallArray));
Console.WriteLine(string.Join(" ", largeArray));
// Output:
// 1 2 100 4 5
// 10 20 0 40 50
```

You can ref assign the result of a conditional ref expression, use it as a reference return or pass it as a `ref`, `out`, `in`, or `ref readonly` method parameter. You can also assign to the result of a conditional ref expression, as the preceding example shows.

The syntax for a conditional ref expression is as follows:

C#

```
condition ? ref consequent : ref alternative
```

Like the conditional operator, a conditional ref expression evaluates only one of the two expressions: either `consequent` or `alternative`.

In a conditional ref expression, the type of `consequent` and `alternative` must be the same. Conditional ref expressions aren't target-typed.

Conditional operator and an `if` statement

Use of the conditional operator instead of an if statement might result in more concise code in cases when you need conditionally to compute a value. The following example demonstrates two ways to classify an integer as negative or nonnegative:

C#

```
int input = new Random().Next(-5, 5);

string classify;
if (input >= 0)
{
    classify = "nonnegative";
}
else
{
    classify = "negative";
```

```
}

classify = (input >= 0) ? "nonnegative" : "negative";
```

Operator overloadability

A user-defined type can't overload the conditional operator.

C# language specification

For more information, see the Conditional operator section of the C# language specification.

Specifications for newer features are:

- Target-typed conditional expression

See also

- Simplify conditional expression (style rule IDE0075)
- C# operators and expressions
- if statement
- ?. and ?[] operators
- ?? and ??= operators
- ref keyword

! (null-forgiving) operator (C# reference)

Article • 12/02/2022

The unary postfix `!` operator is the null-forgiving, or null-suppression, operator. In an enabled nullable annotation context, you use the null-forgiving operator to suppress all nullable warnings for the preceding expression. The unary prefix `!` operator is the logical negation operator. The null-forgiving operator has no effect at run time. It only affects the compiler's static flow analysis by changing the null state of the expression. At run time, expression `x!` evaluates to the result of the underlying expression `x`.

For more information about the nullable reference types feature, see Nullable reference types.

Examples

One of the use cases of the null-forgiving operator is in testing the argument validation logic. For example, consider the following class:

C#

```
#nullable enable
public class Person
{
    public Person(string name) => Name = name ?? throw new
ArgumentNullException(nameof(name));

    public string Name { get; }
}
```

Using the MSTest test framework, you can create the following test for the validation logic in the constructor:

C#

```
[TestMethod, ExpectedException(typeof(ArgumentNullException))]
public void NullNameShouldThrowTest()
{
    var person = new Person(null!);
}
```

Without the null-forgiving operator, the compiler generates the following warning for the preceding code: `Warning CS8625: Cannot convert null literal to non-nullable`

`reference type`. By using the null-forgiving operator, you inform the compiler that passing `null` is expected and shouldn't be warned about.

You can also use the null-forgiving operator when you definitely know that an expression can't be `null` but the compiler doesn't manage to recognize that. In the following example, if the `IsValid` method returns `true`, its argument isn't `null` and you can safely dereference it:

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p!.Name}");
    }
}

public static bool IsValid(Person? person)
    => person is not null && person.Name is not null;
```

Without the null-forgiving operator, the compiler generates the following warning for the `p.Name` code: `Warning CS8602: Dereference of a possibly null reference`.

If you can modify the `IsValid` method, you can use the NotNullWhen attribute to inform the compiler that an argument of the `IsValid` method can't be `null` when the method returns `true`:

C#

```
public static void Main()
{
    Person? p = Find("John");
    if (IsValid(p))
    {
        Console.WriteLine($"Found {p.Name}");
    }
}

public static bool IsValid([NotNullWhen(true)] Person? person)
    => person is not null && person.Name is not null;
```

In the preceding example, you don't need to use the null-forgiving operator because the compiler has enough information to find out that `p` can't be `null` inside the `if` statement. For more information about the attributes that allow you to provide

additional information about the null state of a variable, see Upgrade APIs with attributes to define null expectations.

C# language specification

For more information, see The null-forgiving operator section of the draft of the nullable reference types specification.

See also

- Remove unnecessary suppression operator (style rule IDE0080)
- C# operators and expressions
- Tutorial: Design with nullable reference types

?? and ??= operators - the null-coalescing operators

The null-coalescing operator `??` returns the value of its left-hand operand if it isn't `null`; otherwise, it evaluates the right-hand operand and returns its result. The `??` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null. The null-coalescing assignment operator `??=` assigns the value of its right-hand operand to its left-hand operand only if the left-hand operand evaluates to `null`. The `??=` operator doesn't evaluate its right-hand operand if the left-hand operand evaluates to non-null.

C#

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

Console.WriteLine((numbers is null)); // expected: true
// if numbers is null, initialize it. Then, add 5 to numbers
(numbers ??= new List<int>()).Add(5);
Console.WriteLine(string.Join(" ", numbers));  // output: 5
Console.WriteLine((numbers is null)); // expected: false


Console.WriteLine((a is null)); // expected: true
Console.WriteLine((a ?? 3)); // expected: 3 since a is still null
// if a is null then assign 0 to a and add a to the list
numbers.Add(a ??= 0);
Console.WriteLine((a is null)); // expected: false
Console.WriteLine(string.Join(" ", numbers));  // output: 5 0
Console.WriteLine(a);  // output: 0
```

The left-hand operand of the `??=` operator must be a variable, a property, or an indexer element.

The type of the left-hand operand of the `??` and `??=` operators can't be a non-nullable value type. In particular, you can use the null-coalescing operators with unconstrained type parameters:

C#

```
private static void Display<T>(T a, T backup)
{
    Console.WriteLine(a ?? backup);
}
```

The null-coalescing operators are right-associative. That is, expressions of the form

C#

```
a ?? b ?? c
d ??= e ??= f
```

are evaluated as

C#

```
a ?? (b ?? c)
d ??= (e ??= f)
```

Examples

The `??` and `??=` operators can be useful in the following scenarios:

- In expressions with the null-conditional operators ?. and ?[], you can use the `??` operator to provide an alternative expression to evaluate in case the result of the expression with null-conditional operations is `null`:

  C#

  ```
  double SumNumbers(List<double[]> setsOfNumbers, int indexOfSetToSum)
  {
      return setsOfNumbers?[indexOfSetToSum]?.Sum() ?? double.NaN;
  }

  var sum = SumNumbers(null, 0);
  Console.WriteLine(sum);  // output: NaN
  ```

- When you work with nullable value types and need to provide a value of an underlying value type, use the `??` operator to specify the value to provide in case a nullable type value is `null`:

  C#

  ```
  int? a = null;
  int b = a ?? -1;
  Console.WriteLine(b);  // output: -1
  ```

  Use the Nullable<T>.GetValueOrDefault() method if the value to be used when a nullable type value is `null` should be the default value of the underlying value type.

- You can use a throw expression as the right-hand operand of the `??` operator to make the argument-checking code more concise:

```C#
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), "Name
cannot be null");
}
```

The preceding example also demonstrates how to use expression-bodied members to define a property.

- You can use the `??=` operator to replace the code of the form

```C#
if (variable is null)
{
    variable = expression;
}
```

with the following code:

```C#
variable ??= expression;
```

Operator overloadability

The operators `??` and `??=` can't be overloaded.

C# language specification

For more information about the `??` operator, see The null coalescing operator section of the C# language specification.

For more information about the `??=` operator, see the Compound assignment section of the C# language specification.

See also

- Null check can be simplified (IDE0029, IDE0030, and IDE0270)
- C# operators and expressions
- ?. and ?[] operators

- ?: operator

Last updated on 11/01/2025

Lambda expression (`=>`) operator defines a lambda expression

Article • 02/19/2025

The `=>` token is supported in two forms: as the lambda operator and as a separator of a member name and the member implementation in an expression body definition.

Lambda operator

In lambda expressions, the lambda operator `=>` separates the input parameters on the left side from the lambda body on the right side.

The following example uses the LINQ feature with method syntax to demonstrate the usage of lambda expressions:

C#

```
string[] words = { "bot", "apple", "apricot" };
int minimalLength = words
  .Where(w => w.StartsWith("a"))
  .Min(w => w.Length);
Console.WriteLine(minimalLength);   // output: 5

int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (interim, next) => interim * next);
Console.WriteLine(product);   // output: 280
```

Input parameters of a lambda expression are strongly typed at compile time. When the compiler can infer the types of input parameters, like in the preceding example, you can omit type declarations. If you need to specify the type of input parameters, you must do that for each parameter, as the following example shows:

C#

```
int[] numbers = { 4, 7, 10 };
int product = numbers.Aggregate(1, (int interim, int next) => interim *
next);
Console.WriteLine(product);   // output: 280
```

The following example shows how to define a lambda expression without input parameters:

C#

```
Func<string> greet = () => "Hello, World!";
Console.WriteLine(greet());
```

For more information, see Lambda expressions.

Expression body definition

An expression body definition has the following general syntax:

C#

```
member => expression;
```

Where `expression` is a valid expression. The return type of `expression` must be implicitly convertible to the member's return type. If the member:

- Has a `void` return type or
- Is a:
  - Constructor
  - Finalizer
  - Property or indexer `set` accessor

`expression` must be a *statement expression*. Because the expression's result is discarded, the return type of that expression can be any type.

The following example shows an expression body definition for a `Person.ToString` method:

C#

```
public override string ToString() => $"{fname} {lname}".Trim();
```

It's a shorthand version of the following method definition:

C#

```
public override string ToString()
{
   return $"{fname} {lname}".Trim();
}
```

You can create expression body definitions for methods, operators, read-only properties, constructors, finalizers, and property and indexer accessors. For more information, see Expression-bodied members.

Operator overloadability

The `=>` operator can't be overloaded.

C# language specification

For more information about the lambda operator, see the Anonymous function expressions section of the C# language specification.

See also

- C# operators and expressions

:: operator - the namespace alias operator

Article • 04/12/2023

Use the namespace alias qualifier `::` to access a member of an aliased namespace. You can use the `::` qualifier only between two identifiers. The left-hand identifier can be one of a namespace alias, an extern alias, or the `global` alias. For example:

- A namespace alias created with a using alias directive:

  C#

  ```
  using forwinforms = System.Drawing;
  using forwpf = System.Windows;

  public class Converters
  {
      public static forwpf::Point Convert(forwinforms::Point point) =>
  new forwpf::Point(point.X, point.Y);
  }
  ```

- An extern alias.

- The `global` alias, which is the global namespace alias. The global namespace is the namespace that contains namespaces and types that aren't declared inside a named namespace. When used with the `::` qualifier, the `global` alias always references the global namespace, even if there's the user-defined `global` namespace alias.

  The following example uses the `global` alias to access the .NET System namespace, which is a member of the global namespace. Without the `global` alias, the user-defined `System` namespace, which is a member of the `MyCompany.MyProduct` namespace, would be accessed:

  C#

  ```
  namespace MyCompany.MyProduct.System
  {
      class Program
      {
          static void Main() => global::System.Console.WriteLine("Using
  global alias");
      }
  ```

```
    class Console
    {
        string Suggestion => "Consider renaming this class";
    }
}
```

> **Note**
>
> The `global` keyword is the global namespace alias only when it's the left-hand identifier of the `::` qualifier.

You can also use the . token to access a member of an aliased namespace. However, the `.` token is also used to access a type member. The `::` qualifier ensures that its left-hand identifier always references a namespace alias, even if there exists a type or namespace with the same name.

C# language specification

For more information, see the Namespace alias qualifiers section of the C# language specification.

See also

- C# operators and expressions

await operator - asynchronously await for a task to complete

Article • 07/01/2024

The `await` operator suspends evaluation of the enclosing async method until the asynchronous operation represented by its operand completes. When the asynchronous operation completes, the `await` operator returns the result of the operation, if any. When the `await` operator is applied to the operand that represents an already completed operation, it returns the result of the operation immediately without suspension of the enclosing method. The `await` operator doesn't block the thread that evaluates the async method. When the `await` operator suspends the enclosing async method, the control returns to the caller of the method.

In the following example, the HttpClient.GetByteArrayAsync method returns the `Task<byte[]>` instance, which represents an asynchronous operation that produces a byte array when it completes. Until the operation completes, the `await` operator suspends the `DownloadDocsMainPageAsync` method. When `DownloadDocsMainPageAsync` gets suspended, control is returned to the `Main` method, which is the caller of `DownloadDocsMainPageAsync`. The `Main` method executes until it needs the result of the asynchronous operation performed by the `DownloadDocsMainPageAsync` method. When GetByteArrayAsync gets all the bytes, the rest of the `DownloadDocsMainPageAsync` method is evaluated. After that, the rest of the `Main` method is evaluated.

C#

```
public class AwaitOperator
{
    public static async Task Main()
    {
        Task<int> downloading = DownloadDocsMainPageAsync();
        Console.WriteLine($"{nameof(Main)}: Launched downloading.");

        int bytesLoaded = await downloading;
        Console.WriteLine($"{nameof(Main)}: Downloaded {bytesLoaded}
bytes.");
    }

    private static async Task<int> DownloadDocsMainPageAsync()
    {
        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: About to
start downloading.");

        var client = new HttpClient();
        byte[] content = await
```

```
client.GetByteArrayAsync("https://learn.microsoft.com/en-us/");

        Console.WriteLine($"{nameof(DownloadDocsMainPageAsync)}: Finished
downloading.");
        return content.Length;
    }
}
// Output similar to:
// DownloadDocsMainPageAsync: About to start downloading.
// Main: Launched downloading.
// DownloadDocsMainPageAsync: Finished downloading.
// Main: Downloaded 27700 bytes.
```

The operand of an `await` expression must provide for notification when a task completes. In general, a delegate is invoked when the task completes, either successfully or unsuccessfully. The await section of the C# language spec provides the details on how these notifications are implemented.

The preceding example uses the async Main method. For more information, see the await operator in the Main method section.

> **Note**
>
> For an introduction to asynchronous programming, see Asynchronous programming with async and await. Asynchronous programming with `async` and `await` follows the task-based asynchronous pattern.

You can use the `await` operator only in a method, lambda expression, or anonymous method that is modified by the async keyword. Within an async method, you can't use the `await` operator in the body of a synchronous local function, inside the block of a lock statement, and in an unsafe context.

The operand of the `await` operator is usually of one of the following .NET types: Task, Task<TResult>, ValueTask, or ValueTask<TResult>. However, any awaitable expression can be the operand of the `await` operator. For more information, see the Awaitable expressions section of the C# language specification.

The type of expression `await t` is `TResult` if the type of expression `t` is Task<TResult> or ValueTask<TResult>. If the type of `t` is Task or ValueTask, the type of `await t` is `void`. In both cases, if `t` throws an exception, `await t` rethrows the exception.

Asynchronous streams and disposables

You use the `await foreach` statement to consume an asynchronous stream of data. For more information, see the foreach statement section of the Iteration statements article.

You use the `await using` statement to work with an asynchronously disposable object, that is, an object of a type that implements an IAsyncDisposable interface. For more information, see the Using async disposable section of the Implement a DisposeAsync method article.

await operator in the Main method

The Main method, which is the application entry point, can return `Task` or `Task<int>`, enabling it to be async so you can use the `await` operator in its body. In earlier C# versions, to ensure that the `Main` method waits for the completion of an asynchronous operation, you can retrieve the value of the Task<TResult>.Result property of the Task<TResult> instance that is returned by the corresponding async method. For asynchronous operations that don't produce a value, you can call the Task.Wait method. For information about how to select the language version, see C# language versioning.

C# language specification

For more information, see the Await expressions section of the C# language specification.

See also

- C# operators and expressions
- async
- Task asynchronous programming model
- Asynchronous programming
- Walkthrough: accessing the Web by using async and await
- Tutorial: Generate and consume async streams
- .NET blog: How async/await really works in C#

default value expressions - produce the default value

Article • 03/30/2024

A default value expression produces the default value of a type. There are two kinds of default value expressions: the default operator call and a default literal.

You also use the `default` keyword as the default case label within a switch statement.

default operator

The argument to the `default` operator must be the name of a type or a type parameter, as the following example shows:

C#

```
Console.WriteLine(default(int));  // output: 0
Console.WriteLine(default(object) is null);  // output: True

void DisplayDefaultOf<T>()
{
    var val = default(T);
    Console.WriteLine($"Default value of {typeof(T)} is {(val == null ?
"null" : val.ToString())}.");
}

DisplayDefaultOf<int?>();
DisplayDefaultOf<System.Numerics.Complex>();
DisplayDefaultOf<System.Collections.Generic.List<int>>();
// Output:
// Default value of System.Nullable`1[System.Int32] is null.
// Default value of System.Numerics.Complex is (0, 0).
// Default value of System.Collections.Generic.List`1[System.Int32] is null.
```

default literal

You can use the `default` literal to produce the default value of a type when the compiler can infer the expression type. The `default` literal expression produces the same value as the `default(T)` expression where `T` is the inferred type. You can use the `default` literal in any of the following cases:

- In the assignment or initialization of a variable.
- In the declaration of the default value for an optional method parameter.

- In a method call to provide an argument value.
- In a return statement or as an expression in an expression-bodied member.

The following example shows the usage of the `default` literal:

C#

```
T[] InitializeArray<T>(int length, T initialValue = default)
{
    if (length < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(length), "Array length
must be nonnegative.");
    }

    var array = new T[length];
    for (var i = 0; i < length; i++)
    {
        array[i] = initialValue;
    }
    return array;
}

void Display<T>(T[] values) => Console.WriteLine($"[ {string.Join(", ",
values)} ]");

Display(InitializeArray<int>(3));  // output: [ 0, 0, 0 ]
Display(InitializeArray<bool>(4, default));  // output: [ False, False,
False, False ]

System.Numerics.Complex fillValue = default;
Display(InitializeArray(3, fillValue));  // output: [ (0, 0), (0, 0), (0, 0)
]
```

> **Tip**
>
> Use .NET style rule **IDE0034** to specify a preference on the use of the `default` literal in your codebase.

C# language specification

For more information, see the Default value expressions section of the C# language specification.

See also

- C# operators and expressions
- Default values of C# types
- Generics in .NET

delegate operator

Article • 03/30/2024

The `delegate` operator creates an anonymous method that can be converted to a delegate type. An anonymous method can be converted to types such as System.Action and System.Func<TResult> types used as arguments to many methods.

C#

```
Func<int, int, int> sum = delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(3, 4));  // output: 7
```

> **Note**
>
> Lambda expressions provide a more concise and expressive way to create an anonymous function. Use the **=> operator** to construct a lambda expression:
>
> C#
>
> ```
> Func<int, int, int> sum = (a, b) => a + b;
> Console.WriteLine(sum(3, 4));  // output: 7
> ```
>
> For more information about features of lambda expressions, for example, capturing outer variables, see **Lambda expressions**.

When you use the `delegate` operator, you might omit the parameter list. If you do that, the created anonymous method can be converted to a delegate type with any list of parameters, as the following example shows:

C#

```
Action greet = delegate { Console.WriteLine("Hello!"); };
greet();

Action<int, double> introduce = delegate { Console.WriteLine("This is
world!"); };
introduce(42, 2.7);

// Output:
// Hello!
// This is world!
```

That's the only functionality of anonymous methods that isn't supported by lambda expressions. In all other cases, a lambda expression is a preferred way to write inline code. You can use discards to specify two or more input parameters of an anonymous method that aren't used by the method:

```
C#
```

```
Func<int, int, int> constant = delegate (int _, int _) { return 42; };
Console.WriteLine(constant(3, 4));  // output: 42
```

For backwards compatibility, if only a single parameter is named `_`, `_` is treated as the name of that parameter within an anonymous method.

You can use the `static` modifier at the declaration of an anonymous method:

```
C#
```

```
Func<int, int, int> sum = static delegate (int a, int b) { return a + b; };
Console.WriteLine(sum(10, 4));  // output: 14
```

A static anonymous method can't capture local variables or instance state from enclosing scopes.

You also use the `delegate` keyword to declare a delegate type.

Beginning with C# 11, the compiler may cache the delegate object created from a method group. Consider the following method:

```
C#
```

```
static void StaticFunction() { }
```

When you assign the method group to a delegate, the compiler will cache the delegate:

```
C#
```

```
Action a = StaticFunction;
```

Before C# 11, you'd need to use a lambda expression to reuse a single delegate object:

```
C#
```

```
Action a = () => StaticFunction();
```

C# language specification

For more information, see the Anonymous function expressions section of the C# language specification.

See also

- C# operators and expressions
- => operator

The `is` operator (C# reference)

Article • 02/18/2025

The `is` operator checks if the result of an expression is compatible with a given type. For information about the type-testing `is` operator, see the is operator section of the Type-testing and cast operators article. You can also use the `is` operator to match an expression against a pattern, as the following example shows:

```
C#

static bool IsFirstFridayOfOctober(DateTime date) =>
    date is { Month: 10, Day: <=7, DayOfWeek: DayOfWeek.Friday };
```

In the preceding example, the `is` operator matches an expression against a property pattern with nested constant and relational patterns.

The `is` operator can be useful in the following scenarios:

- To check the run-time type of an expression, as the following example shows:

  ```
  C#

  int i = 34;
  object iBoxed = i;
  int? jNullable = 42;
  if (iBoxed is int a && jNullable is int b)
  {
      Console.WriteLine(a + b);  // output 76
  }
  ```

  The preceding example shows the use of a declaration pattern.

- To check for `null`, as the following example shows:

  ```
  C#

  if (input is null)
  {
      return;
  }
  ```

  When you match an expression against `null`, the compiler guarantees that no user-overloaded `==` or `!=` operator is invoked.

- You can use a negation pattern to do a non-null check, as the following example shows:

  C#

  ```
  if (result is not null)
  {
      Console.WriteLine(result.ToString());
  }
  ```

- Beginning with C# 11, you can use list patterns to match elements of a list or array. The following code checks arrays for integer values in expected positions:

  C#

  ```
  int[] empty = [];
  int[] one = [1];
  int[] odd = [1, 3, 5];
  int[] even = [2, 4, 6];
  int[] fib = [1, 1, 2, 3, 5];

  Console.WriteLine(odd is [1, _, 2, ..]);   // false
  Console.WriteLine(fib is [1, _, 2, ..]);   // true
  Console.WriteLine(fib is [_, 1, 2, 3, ..]);     // true
  Console.WriteLine(fib is [.., 1, 2, 3, _ ]);     // true
  Console.WriteLine(even is [2, _, 6]);     // true
  Console.WriteLine(even is [2, .., 6]);    // true
  Console.WriteLine(odd is [.., 3, 5]); // true
  Console.WriteLine(even is [.., 3, 5]); // false
  Console.WriteLine(fib is [.., 3, 5]); // true
  ```

> **Note**
>
> For the complete list of patterns supported by the `is` operator, see Patterns.

C# language specification

For more information, see The is operator section of the C# language specification and Pattern matching.

See also

- C# operators and expressions
- Patterns

- Tutorial: Use pattern matching to build type-driven and data-driven algorithms
- Type-testing and cast operators

nameof expression (C# reference)

Article • 02/25/2025

A `nameof` expression produces the name of a variable, type, or member as the string constant. A `nameof` expression is evaluated at compile time and has no effect at run time. When the operand is a type or a namespace, the produced name isn't fully qualified. The following example shows the use of a `nameof` expression:

C#

```
Console.WriteLine(nameof(System.Collections.Generic));  // output: Generic
Console.WriteLine(nameof(List<int>));  // output: List
Console.WriteLine(nameof(List<>)); // output: List
Console.WriteLine(nameof(List<int>.Count));  // output: Count
Console.WriteLine(nameof(List<int>.Add));  // output: Add

List<int> numbers = new List<int>() { 1, 2, 3 };
Console.WriteLine(nameof(numbers));  // output: numbers
Console.WriteLine(nameof(numbers.Count));  // output: Count
Console.WriteLine(nameof(numbers.Add));  // output: Add
```

The preceding example using `List<>` is supported in C# 14 and later. You can use a `nameof` expression to make the argument-checking code more maintainable:

C#

```
public string Name
{
    get => name;
    set => name = value ?? throw new ArgumentNullException(nameof(value), $"
{nameof(Name)} cannot be null");
}
```

Beginning with C# 11, you can use a `nameof` expression with a method parameter inside an attribute on a method or its parameter. The following code shows how to do that for an attribute on a method, a local function, and the parameter of a lambda expression:

C#

```
[ParameterString(nameof(msg))]
public static void Method(string msg)
{
    [ParameterString(nameof(T))]
    void LocalFunction<T>(T param) { }

    var lambdaExpression = ([ParameterString(nameof(aNumber))] int aNumber)
```

```
=> aNumber.ToString();
}
```

A `nameof` expression with a parameter is useful when you use the nullable analysis attributes or the CallerArgumentExpression attribute.

When the operand is a verbatim identifier, the `@` character isn't part of the name, as the following example shows:

```C#
var @new = 5;
Console.WriteLine(nameof(@new));  // output: new
```

C# language specification

For more information, see the Nameof expressions section of the C# language specification, and the C# 11 - Extended nameof scope feature specification.

See also

- C# operators and expressions
- Convert typeof to nameof (style rule IDE0082)

new operator - The `new` operator creates a new instance of a type

Article • 11/14/2023

The `new` operator creates a new instance of a type. You can also use the `new` keyword as a member declaration modifier or a generic type constraint.

Constructor invocation

To create a new instance of a type, you typically invoke one of the constructors of that type using the `new` operator:

C#

```
var dict = new Dictionary<string, int>();
dict["first"] = 10;
dict["second"] = 20;
dict["third"] = 30;

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

You can use an object or collection initializer with the `new` operator to instantiate and initialize an object in one statement, as the following example shows:

C#

```
var dict = new Dictionary<string, int>
{
    ["first"] = 10,
    ["second"] = 20,
    ["third"] = 30
};

Console.WriteLine(string.Join("; ", dict.Select(entry => $"{entry.Key}:
{entry.Value}")));
// Output:
// first: 10; second: 20; third: 30
```

Target-typed `new`

Constructor invocation expressions are target-typed. That is, if a target type of an expression is known, you can omit a type name, as the following example shows:

C#

```
List<int> xs = new();
List<int> ys = new(capacity: 10_000);
List<int> zs = new() { Capacity = 20_000 };

Dictionary<int, List<int>> lookup = new()
{
    [1] = new() { 1, 2, 3 },
    [2] = new() { 5, 8, 3 },
    [5] = new() { 1, 0, 4 }
};
```

As the preceding example shows, you always use parentheses in a target-typed `new` expression.

If a target type of a `new` expression is unknown (for example, when you use the var keyword), you must specify a type name.

Array creation

You also use the `new` operator to create an array instance, as the following example shows:

C#

```
var numbers = new int[3];
numbers[0] = 10;
numbers[1] = 20;
numbers[2] = 30;
Console.WriteLine(string.Join(", ", numbers));
// Output:
// 10, 20, 30
```

Use array initialization syntax to create an array instance and populate it with elements in one statement. The following example shows various ways how you can do that:

C#

```
var a = new int[3] { 10, 20, 30 };
var b = new int[] { 10, 20, 30 };
var c = new[] { 10, 20, 30 };
Console.WriteLine(c.GetType());  // output: System.Int32[]
```

For more information about arrays, see Arrays.

Instantiation of anonymous types

To create an instance of an anonymous type, use the `new` operator and object initializer syntax:

```
C#
```

```
var example = new { Greeting = "Hello", Name = "World" };
Console.WriteLine($"{example.Greeting}, {example.Name}!");
// Output:
// Hello, World!
```

Destruction of type instances

You don't have to destroy earlier created type instances. Instances of both reference and value types are destroyed automatically. Instances of value types are destroyed as soon as the context that contains them is destroyed. Instances of reference types are destroyed by the garbage collector at some unspecified time after the last reference to them is removed.

For type instances that contain unmanaged resources, for example, a file handle, it's recommended to employ deterministic clean-up to ensure that the resources they contain are released as soon as possible. For more information, see the System.IDisposable API reference and the using statement article.

Operator overloadability

A user-defined type can't overload the `new` operator.

C# language specification

For more information, see The new operator section of the C# language specification.

For more information about a target-typed `new` expression, see the feature proposal note.

See also

- C# operators and expressions
- Object and collection initializers

sizeof operator - determine the memory needs for a given type

Article • 02/07/2025

The `sizeof` operator returns the number of bytes occupied by a variable of a given type. In safe code, the argument to the `sizeof` operator must be the name of an unmanaged type or a type parameter that is constrained to be an unmanaged type. Unmanaged types include all numeric types, enum types, and tuple and struct types where all members are unmanaged types.

The expressions presented in the following table are evaluated in compile time to the corresponding constant values and don't require an unsafe context:

Expand table

Expression	Constant value
`sizeof(sbyte)`	1
`sizeof(byte)`	1
`sizeof(short)`	2
`sizeof(ushort)`	2
`sizeof(int)`	4
`sizeof(uint)`	4
`sizeof(long)`	8
`sizeof(ulong)`	8
`sizeof(char)`	2
`sizeof(float)`	4
`sizeof(double)`	8
`sizeof(decimal)`	16
`sizeof(bool)`	1

In unsafe code, the argument to `sizeof` can include pointer types and managed types, including unconstrained type parameters. Examples include `object` and `string`.

The following example demonstrates the usage of the `sizeof` operator:

C#

```
public struct Point
{
    public Point(byte tag, double x, double y) => (Tag, X, Y) = (tag, x, y);

    public byte Tag { get; }
    public double X { get; }
    public double Y { get; }
}

public class SizeOfOperator
{
    public static void Main()
    {
        Console.WriteLine(sizeof(byte));  // output: 1
        Console.WriteLine(sizeof(double));  // output: 8

        DisplaySizeOf<Point>();  // output: Size of Point is 24
        DisplaySizeOf<decimal>();  // output: Size of System.Decimal is 16

        unsafe
        {
            Console.WriteLine(sizeof(Point*));  // output: 8
        }
    }

    static unsafe void DisplaySizeOf<T>() where T : unmanaged
    {
        Console.WriteLine($"Size of {typeof(T)} is {sizeof(T)}");
    }
}
```

The `sizeof` operator returns the number of bytes allocated by the common language runtime in managed memory. For struct types, that value includes any padding, as the preceding example demonstrates. The result of the `sizeof` operator might differ from the result of the Marshal.SizeOf method, which returns the size of a type in *unmanaged* memory.

In unsafe code, when the argument is a managed type, the `sizeof` operator returns the size of a reference, not the number of bytes allocated for an instance of that type.

> **Important**
>
> The value returned by `sizeof` can differ from the result of Marshal.SizeOf(Object), which returns the size of the type in unmanaged memory.

C# language specification

For more information, see The sizeof operator section of the C# language specification.

See also

- C# operators and expressions
- Pointer related operators
- Pointer types
- Memory and span-related types
- Generics in .NET

stackalloc expression (C# reference)

Article • 07/10/2024

A `stackalloc` expression allocates a block of memory on the stack. A stack-allocated memory block created during the method execution is automatically discarded when that method returns. You can't explicitly free the memory allocated with `stackalloc`. A stack allocated memory block isn't subject to garbage collection and doesn't have to be pinned with a fixed statement.

You can assign the result of a `stackalloc` expression to a variable of one of the following types:

- System.Span<T> or System.ReadOnlySpan<T>, as the following example shows:

  C#

  ```
  int length = 3;
  Span<int> numbers = stackalloc int[length];
  for (var i = 0; i < length; i++)
  {
      numbers[i] = i;
  }
  ```

  You don't have to use an unsafe context when you assign a stack allocated memory block to a Span<T> or ReadOnlySpan<T> variable.

  When you work with those types, you can use a `stackalloc` expression in conditional or assignment expressions, as the following example shows:

  C#

  ```
  int length = 1000;
  Span<byte> buffer = length <= 1024 ? stackalloc byte[length] : new
  byte[length];
  ```

  You can use a `stackalloc` expression or a collection expression inside other expressions whenever a Span<T> or ReadOnlySpan<T> variable is allowed, as the following example shows:

  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
  ```

```
Span<int> numbers2 = [1, 2, 3, 4, 5, 6];
var ind2 = numbers2.IndexOfAny([2, 4, 6, 8]);
Console.WriteLine(ind2);  // output: 1
```

> **Note**
>
> We recommend using Span<T> or ReadOnlySpan<T> types to work with stack allocated memory whenever possible.

- A pointer type, as the following example shows:

```
C#

unsafe
{
    int length = 3;
    int* numbers = stackalloc int[length];
    for (var i = 0; i < length; i++)
    {
        numbers[i] = i;
    }
}
```

As the preceding example shows, you must use an `unsafe` context when you work with pointer types.

In the case of pointer types, you can use a `stackalloc` expression only in a local variable declaration to initialize the variable.

The amount of memory available on the stack is limited. If you allocate too much memory on the stack, a StackOverflowException is thrown. To avoid that, follow the rules below:

- Limit the amount of memory you allocate with `stackalloc`. For example, if the intended buffer size is below a certain limit, you allocate the memory on the stack; otherwise, use an array of the required length, as the following code shows:

```
C#

const int MaxStackLimit = 1024;
Span<byte> buffer = inputLength <= MaxStackLimit ? stackalloc
byte[MaxStackLimit] : new byte[inputLength];
```

> **Note**
>
> Because the amount of memory available on the stack depends on the environment in which the code is executed, be conservative when you define the actual limit value.

- Avoid using `stackalloc` inside loops. Allocate the memory block outside a loop and reuse it inside the loop.

The content of the newly allocated memory is undefined. You should initialize it, either with a `stackalloc` initializer, or a method like Span<T>.Clear before it's used.

> **Important**
>
> Not initializing memory allocated by `stackalloc` is an important difference from the `new` operator. Memory allocated using the `new` operator is initialized to the 0 bit pattern.

You can use array initializer syntax to define the content of the newly allocated memory. The following example demonstrates various ways to do that:

C#

```
Span<int> first = stackalloc int[3] { 1, 2, 3 };
Span<int> second = stackalloc int[] { 1, 2, 3 };
ReadOnlySpan<int> third = stackalloc[] { 1, 2, 3 };

// Using collection expressions:
Span<int> fourth = [1, 2, 3];
ReadOnlySpan<int> fifth = [1, 2, 3];
```

In expression `stackalloc T[E]`, `T` must be an unmanaged type and `E` must evaluate to a non-negative int value. When you use the collection expression syntax to initialize the span, the compiler may use stack allocated storage for a span if it won't violate ref safety.

Security

The use of `stackalloc` automatically enables buffer overrun detection features in the common language runtime (CLR). If a buffer overrun is detected, the process is

terminated as quickly as possible to minimize the chance that malicious code is executed.

C# language specification

For more information, see the Stack allocation section of the C# language specification.

See also

- C# operators and expressions
- Pointer related operators
- Pointer types
- Memory and span-related types
- Dos and Don'ts of stackalloc

switch expression - pattern matching expressions using the `switch` keyword

Article • 12/02/2022

You use the `switch` expression to evaluate a single expression from a list of candidate expressions based on a pattern match with an input expression. For information about the `switch` statement that supports `switch`-like semantics in a statement context, see the switch statement section of the Selection statements article.

The following example demonstrates a `switch` expression, which converts values of an enum representing visual directions in an online map to the corresponding cardinal directions:

C#

```
public static class SwitchExample
{
    public enum Direction
    {
        Up,
        Down,
        Right,
        Left
    }

    public enum Orientation
    {
        North,
        South,
        East,
        West
    }

    public static Orientation ToOrientation(Direction direction) =>
direction switch
    {
        Direction.Up    => Orientation.North,
        Direction.Right => Orientation.East,
        Direction.Down  => Orientation.South,
        Direction.Left  => Orientation.West,
        _ => throw new ArgumentOutOfRangeException(nameof(direction), $"Not
expected direction value: {direction}"),
    };

    public static void Main()
    {
        var direction = Direction.Right;
        Console.WriteLine($"Map view direction is {direction}");
```

```
        Console.WriteLine($"Cardinal orientation is
{ToOrientation(direction)}");
        // Output:
        // Map view direction is Right
        // Cardinal orientation is East
    }
}
```

The preceding example shows the basic elements of a `switch` expression:

- An expression followed by the `switch` keyword. In the preceding example, it's the `direction` method parameter.
- The *`switch` expression arms*, separated by commas. Each `switch` expression arm contains a *pattern*, an optional *case guard*, the `=>` token, and an *expression*.

At the preceding example, a `switch` expression uses the following patterns:

- A constant pattern: to handle the defined values of the `Direction` enumeration.
- A discard pattern: to handle any integer value that doesn't have the corresponding member of the `Direction` enumeration (for example, `(Direction)10`). That makes the `switch` expression exhaustive.

> **Important**
>
> For information about the patterns supported by the `switch` expression and more examples, see **Patterns**.

The result of a `switch` expression is the value of the expression of the first `switch` expression arm whose pattern matches the input expression and whose case guard, if present, evaluates to `true`. The `switch` expression arms are evaluated in text order.

The compiler generates an error when a lower `switch` expression arm can't be chosen because a higher `switch` expression arm matches all its values.

Case guards

A pattern may be not expressive enough to specify the condition for the evaluation of an arm's expression. In such a case, you can use a *case guard*. A *case guard* is another condition that must be satisfied together with a matched pattern. A case guard must be a Boolean expression. You specify a case guard after the `when` keyword that follows a pattern, as the following example shows:

```
```

C#

```
public readonly struct Point
{
    public Point(int x, int y) => (X, Y) = (x, y);

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

static Point Transform(Point point) => point switch
{
    { X: 0, Y: 0 }                    => new Point(0, 0),
    { X: var x, Y: var y } when x < y => new Point(x + y, y),
    { X: var x, Y: var y } when x > y => new Point(x - y, y),
    { X: var x, Y: var y }            => new Point(2 * x, 2 * y),
};
```

The preceding example uses property patterns with nested var patterns.

Non-exhaustive switch expressions

If none of a `switch` expression's patterns matches an input value, the runtime throws an exception. In .NET Core 3.0 and later versions, the exception is a System.Runtime.CompilerServices.SwitchExpressionException. In .NET Framework, the exception is an InvalidOperationException. In most cases, the compiler generates a warning if a `switch` expression doesn't handle all possible input values. List patterns don't generate a warning when all possible inputs aren't handled.

> **Tip**
>
> To guarantee that a `switch` expression handles all possible input values, provide a `switch` expression arm with a **discard pattern**.

C# language specification

For more information, see the switch expression section of the feature proposal note.

See also

- Use switch expression (style rule IDE0066)
- Add missing cases to switch expression (style rule IDE0072)

- C# operators and expressions
- Patterns
- Tutorial: Use pattern matching to build type-driven and data-driven algorithms
- switch statement

true and false operators - treat your objects as a Boolean value

Article • 02/23/2024

The `true` operator returns the bool value `true` to indicate that its operand is definitely true, while the `false` operator returns the `bool` value `true` to indicate that its operand is definitely false.
Note that a type implementing both `true` and `false` operators has to follow these semantics:

- "Is this object true?" resolves to operator `true`. Operator `true` returns `true` if the object is `true`. The answer is "Yes, this object is true".
- "Is this object false?" resolves to operator `false`. Operator `false` returns `true` if the object is `false`. The answer is "Yes, this object is false"

The `true` and `false` operators aren't guaranteed to complement each other. That is, both the `true` and `false` operator might return the `bool` value `false` for the same operand. If a type defines one of these two operators, it must also define the other operator.

> **Tip**
>
> Use the `bool?` type, if you need to support the three-valued logic (for example, when you work with databases that support a three-valued Boolean type). C# provides the `&` and `|` operators that support the three-valued logic with the `bool?` operands. For more information, see the **Nullable Boolean logical operators** section of the **Boolean logical operators** article.

Boolean expressions

A type with the defined `true` operator can be the type of a result of a controlling conditional expression in the if, do, while, and for statements and in the conditional operator ?:. For more information, see the Boolean expressions section of the C# language specification.

User-defined conditional logical operators

If a type with the defined `true` and `false` operators overloads the logical OR operator `|` or the logical AND operator `&` in a certain way, the conditional logical OR operator `||` or conditional logical AND operator `&&`, respectively, can be evaluated for the operands of that type. For more information, see the User-defined conditional logical operators section of the C# language specification.

Example

The following example presents the type that defines both `true` and `false` operators. The type also overloads the logical AND operator `&` in such a way that the `&&` operator also can be evaluated for the operands of that type.

C#

```
public struct LaunchStatus
{
    public static readonly LaunchStatus Green = new LaunchStatus(0);
    public static readonly LaunchStatus Yellow = new LaunchStatus(1);
    public static readonly LaunchStatus Red = new LaunchStatus(2);

    private int status;

    private LaunchStatus(int status)
    {
        this.status = status;
    }

    public static bool operator true(LaunchStatus x) => x == Green || x ==
Yellow;
    public static bool operator false(LaunchStatus x) => x == Red;

    public static LaunchStatus operator &(LaunchStatus x, LaunchStatus y)
    {
        if (x == Red || y == Red || (x == Yellow && y == Yellow))
        {
            return Red;
        }

        if (x == Yellow || y == Yellow)
        {
            return Yellow;
        }

        return Green;
    }

    public static bool operator ==(LaunchStatus x, LaunchStatus y) =>
x.status == y.status;
    public static bool operator !=(LaunchStatus x, LaunchStatus y) => !(x ==
y);
```

```
    public override bool Equals(object obj) => obj is LaunchStatus other &&
this == other;
    public override int GetHashCode() => status;
}

public class LaunchStatusTest
{
    public static void Main()
    {
        LaunchStatus okToLaunch = GetFuelLaunchStatus() &&
GetNavigationLaunchStatus();
        Console.WriteLine(okToLaunch ? "Ready to go!" : "Wait!");
    }

    static LaunchStatus GetFuelLaunchStatus()
    {
        Console.WriteLine("Getting fuel launch status...");
        return LaunchStatus.Red;
    }

    static LaunchStatus GetNavigationLaunchStatus()
    {
        Console.WriteLine("Getting navigation launch status...");
        return LaunchStatus.Yellow;
    }
}
```

Notice the short-circuiting behavior of the `&&` operator. When the `GetFuelLaunchStatus` method returns `LaunchStatus.Red`, the right-hand operand of the `&&` operator isn't evaluated. That is because `LaunchStatus.Red` is definitely false. Then the result of the logical AND doesn't depend on the value of the right-hand operand. The output of the example is as follows:

Console

```
Getting fuel launch status...
Wait!
```

See also

- C# operators and expressions

The `with` expression - Nondestructive mutation creates a new object with modified properties

08/19/2025

A `with` expression produces a copy of its operand with the specified properties and fields modified. You use the object initializer syntax to specify what members to modify and their new values:

C#

```
using System;

public class WithExpressionBasicExample
{
    public record NamedPoint(string Name, int X, int Y);

    public static void Main()
    {
        var p1 = new NamedPoint("A", 0, 0);
        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint {
Name = A, X = 0, Y = 0 }

        var p2 = p1 with { Name = "B", X = 5 };
        Console.WriteLine($"{nameof(p2)}: {p2}");  // output: p2: NamedPoint {
Name = B, X = 5, Y = 0 }

        var p3 = p1 with
            {
                Name = "C",
                Y = 4
            };
        Console.WriteLine($"{nameof(p3)}: {p3}");  // output: p3: NamedPoint {
Name = C, X = 0, Y = 4 }

        Console.WriteLine($"{nameof(p1)}: {p1}");  // output: p1: NamedPoint {
Name = A, X = 0, Y = 0 }

        var apples = new { Item = "Apples", Price = 1.19m };
        Console.WriteLine($"Original: {apples}");  // output: Original: { Item =
Apples, Price = 1.19 }
        var saleApples = apples with { Price = 0.79m };
        Console.WriteLine($"Sale: {saleApples}");  // output: Sale: { Item =
Apples, Price = 0.79 }
    }
}
```

The left-hand operand of a `with` expression can be of a record type. A left-hand operand of a `with` expression can also be of a structure type or an anonymous type.

The result of a `with` expression has the same run-time type as the expression's operand, as the following example shows:

C#

```
using System;

public class InheritanceExample
{
    public record Point(int X, int Y);
    public record NamedPoint(string Name, int X, int Y) : Point(X, Y);

    public static void Main()
    {
        Point p1 = new NamedPoint("A", 0, 0);
        Point p2 = p1 with { X = 5, Y = 3 };
        Console.WriteLine(p2 is NamedPoint);  // output: True
        Console.WriteLine(p2);  // output: NamedPoint { X = 5, Y = 3, Name = A }
    }
}
```

When a member is a reference type, only the reference to a member instance is copied when an operand is copied. Both the copy and original operand have access to the same reference-type instance. The following example demonstrates that behavior:

C#

```
using System;
using System.Collections.Generic;

public class ExampleWithReferenceType
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B, C
```

```
    }
}
```

Custom copy semantics

Any record class type has the *copy constructor*. A *copy constructor* is a constructor with a single parameter of the containing record type. It copies the state of its argument to a new record instance. At evaluation of a `with` expression, the copy constructor gets called to instantiate a new record instance based on an original record. After that, the new instance gets updated according to the specified modifications. By default, the copy constructor is implicit, that is, compiler-generated. If you need to customize the record copy semantics, explicitly declare a copy constructor with the desired behavior. The following example updates the preceding example with an explicit copy constructor. The new copy behavior is to copy list items instead of a list reference when a record is copied:

C#

```
using System;
using System.Collections.Generic;

public class UserDefinedCopyConstructorExample
{
    public record TaggedNumber(int Number, List<string> Tags)
    {
        protected TaggedNumber(TaggedNumber original)
        {
            Number = original.Number;
            Tags = new List<string>(original.Tags);
        }

        public string PrintTags() => string.Join(", ", Tags);
    }

    public static void Main()
    {
        var original = new TaggedNumber(1, new List<string> { "A", "B" });

        var copy = original with { Number = 2 };
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B

        original.Tags.Add("C");
        Console.WriteLine($"Tags of {nameof(copy)}: {copy.PrintTags()}");
        // output: Tags of copy: A, B
    }
}
```

You can't customize the copy semantics for structure types.

> **Important**
>
> In the preceding examples, all properties are independent. None of the properties are computed from other property values. A `with` expression first copies the existing record instance, then modifies any properties or fields specified in the `with` expression. Computed properties in `record` types should be computed on access, not initialized when the instance is created. Otherwise, a property could return the computed value based on the original instance, not the modified copy. For more information, see the language reference article on **record types**.

C# language specification

For more information, see the following sections of the records feature proposal note:

- with expression
- Copy and Clone members

See also

- C# operators and expressions
- Records
- Structure types

Deconstruction expression - Extract properties of fields from a tuple or other user-defined type

Article • 12/21/2024

A *deconstruction expression* extracts data fields from an instance of an object. Each discrete data element is written to a distinct variable, as shown in the following example:

C#

```
var tuple = (X: 1, Y: 2);
var (x, y) = tuple;

Console.WriteLine(x); // output: 1
Console.WriteLine(y); // output: 2
```

The preceding code snippet creates a tuple that has two integer values, `X` and `Y`. The second statement *deconstructs* that tuple and stores the tuple elements in discrete variables `x` and `y`.

Tuple deconstruction

All tuple types support deconstruction expressions. Tuple deconstruction extracts all the tuple's elements. If you only want some of the tuple elements, use a discard for the unused tuple members, as shown in the following example:

C#

```
var tuple2 = (X: 0, Y: 1, Label: "The origin");
var (x2, _, _) = tuple2;
```

In the preceding example, the `Y` and `label` members are discarded. You can specify multiple discards in the same deconstruction expression. You can use discards for all the members of the tuple. The following example is legal, although not useful:

C#

```
var (_, _, _) = tuple2;
```

Record deconstruction

Record types that have a primary constructor support deconstruction for positional parameters. The compiler synthesizes a `Deconstruct` method that extracts the properties synthesized from positional parameters in the primary constructor. The compiler-synthesized `Deconstruction` method doesn't extract properties declared as properties in the record type.

The `record` shown in the following code declares two positional properties, `SquareFeet` and `Address`, along with another property, `RealtorNotes`:

C#

```
public record House(int SquareFeet, string Address)
{
    public required string RealtorNotes { get; set; }
}
```

When you deconstruct a `House` object, all positional properties, and only positional properties, are deconstructed, as shown in the following example:

C#

```
var house = new House(1000, "123 Coder St.")
{
    RealtorNotes = """
    This is a great starter home, with a separate room that's a great home
office setup.
    """
};

var (squareFeet, address) = house;
Console.WriteLine(squareFeet); // output: 1000
Console.WriteLine(address); // output: 123 Coder St.
Console.WriteLine(house.RealtorNotes);
```

You can make use of this behavior to specify which properties of your record types are part of the compiler-synthesized `Deconstruct` method.

Declare `Deconstruct` methods

You can add deconstruction support to any class, struct, or interface you declare. You declare one or `Deconstruct` methods in your type, or as extension methods on that type. A deconstruction expression calls a method `void Deconstruct(out var p1, ..., out var`

`pn)`. The `Deconstruct` method can be either an instance method or an extension method. The type of each parameter in the `Deconstruct` method must match the type of the corresponding argument in the deconstruction expression. The deconstruction expression assigns the value of each argument to the value of the corresponding `out` parameter in the `Deconstruct` method. If multiple `Deconstruct` methods match the deconstruction expression, the compiler reports an error for the ambiguity.

The following code declares a `Point3D` struct that has two `Deconstruct` methods:

C#

```
public struct Point3D
{
    public int X { get; set; }
    public int Y { get; set; }
    public int Z { get; set; }

    public void Deconstruct(out int x, out int y, out int z)
    {
        x = X;
        y = Y;
        z = Z;
    }

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

The first method supports deconstruction expressions that extract all three axis values: `X`, `Y`, and `Z`. The second method supports deconstructing only the planar values: `X` and `Y`. The first method has an *arity* of 3; the second has an arity of 2.

The preceding section described the compiler-synthesized `Deconstruct` method for `record` types with a primary constructor. You can declare more `Deconstruct` methods in record types. These methods can either add other properties, remove some of the default properties, or both. You can also declare a `Deconstruct` that matches the compiler-synthesized signature. If you declare such a `Deconstruct` method, the compiler doesn't synthesize one.

Multiple `Deconstruct` methods are allowed as long as the compiler can determine one unique `Deconstruct` method for a deconstruction expression. Typically, multiple `Deconstruct` methods for the same type have different numbers of parameters. You can also create multiple `Deconstruct` methods that differ by parameter types. However, in

many cases, too many `Deconstruct` methods can lead to ambiguity errors and misleading results.

C# language specification

For more information, see the deconstruction section of the C# Standard.

See also

- C# operators and expressions
- Tuple types
- Records
- Structure types

Operator overloading - predefined unary, arithmetic, equality, and comparison operators

06/13/2025

A user-defined type can overload a predefined C# operator. That is, a type can provide the custom implementation of an operation in case one or both of the operands are of that type. The Overloadable operators section shows which C# operators can be overloaded.

Use the `operator` keyword to declare an operator. An operator declaration must satisfy the following rules:

- It includes a `public` modifier.
- A unary operator has one input parameter. A binary operator has two input parameters. In each case, at least one parameter must have type `T` or `T?` where `T` is the type that contains the operator declaration.
- It includes the `static` modifier, except for the compound assignment operators, such as `+=`.
- The increment (`++`) and decrement (`--`) operators can be implemented as either static or instance methods.

The following example defines a simplified structure to represent a rational number. The structure overloads some of the arithmetic operators:

C#

```
public struct Fraction
{
    private int numerator;
    private int denominator;

    public Fraction(int numerator, int denominator)
    {
        if (denominator == 0)
        {
            throw new ArgumentException("Denominator cannot be zero.",
nameof(denominator));
        }
        this.numerator = numerator;
        this.denominator = denominator;
    }

    public static Fraction operator +(Fraction operand) => operand;
    public static Fraction operator -(Fraction operand) => new Fraction(-
```

```
operand.numerator, operand.denominator);

    public static Fraction operator +(Fraction left, Fraction right)
        => new Fraction(left.numerator * right.denominator + right.numerator *
left.denominator, left.denominator * right.denominator);

    public static Fraction operator -(Fraction left, Fraction right)
        => left + (-right);

    public static Fraction operator *(Fraction left, Fraction right)
        => new Fraction(left.numerator * right.numerator, left.denominator *
right.denominator);

    public static Fraction operator /(Fraction left, Fraction right)
    {
        if (right.numerator == 0)
        {
            throw new DivideByZeroException();
        }
        return new Fraction(left.numerator * right.denominator, left.denominator *
right.numerator);
    }

    // Define increment and decrement to add 1/den, rather than 1/1.
    public static Fraction operator ++(Fraction operand)
        => new Fraction(operand.numerator++, operand.denominator);

    public static Fraction operator --(Fraction operand) =>
        new Fraction(operand.numerator--, operand.denominator);

    public override string ToString() => $"{numerator} / {denominator}";

    // New operators allowed in C# 14:
    public void operator +=(Fraction operand) =>
        (numerator, denominator ) =
        (
            numerator * operand.denominator + operand.numerator * denominator,
            denominator * operand.denominator
        );

    public void operator -=(Fraction operand) =>
        (numerator, denominator) =
        (
            numerator * operand.denominator - operand.numerator * denominator,
            denominator * operand.denominator
        );

    public void operator *=(Fraction operand) =>
        (numerator, denominator) =
        (
            numerator * operand.numerator,
            denominator * operand.denominator
        );

    public void operator /=(Fraction operand)
```

```
    {
        if (operand.numerator == 0)
        {
            throw new DivideByZeroException();
        }
        (numerator, denominator) =
        (
            numerator * operand.denominator,
            denominator * operand.numerator
        );
    }

    public void operator ++() => numerator++;

    public void operator --() => numerator--;
}

public static class OperatorOverloading
{
    public static void Main()
    {
        var a = new Fraction(5, 4);
        var b = new Fraction(1, 2);
        Console.WriteLine(-a);   // output: -5 / 4
        Console.WriteLine(a + b);  // output: 14 / 8
        Console.WriteLine(a - b);  // output: 6 / 8
        Console.WriteLine(a * b);  // output: 5 / 8
        Console.WriteLine(a / b);  // output: 10 / 4
    }
}
```

You could extend the preceding example by defining an implicit conversion from `int` to `Fraction`. Then, overloaded operators would support arguments of those two types. That is, it would become possible to add an integer to a fraction and obtain a fraction as a result.

You also use the `operator` keyword to define a custom type conversion. For more information, see User-defined conversion operators.

Overloadable operators

The following table shows the operators that can be overloaded:

Expand table

Operators	Notes
+x, -x, !x, ~x, ++, --, true, false	The `true` and `false` operators must be overloaded together.

Operators	Notes
x + y, x - y, x * y, x / y, x % y, x & y, x | y, x ^ y, x << y, x >> y, x >>> y	
x == y, x != y, x < y, x > y, x <= y, x >= y	Must be overloaded in pairs as follows: `==` and `!=`, `<` and `>`, `<=` and `>=`.
+=, -=, *=, /=, %=, &=, |=, ^=, <<=, >>=, >>>=	The compound assignment operators can be overloaded in C# 14 and later.

A compound assignment overloaded operator must follow these rules:

- It must include the `public` modifier.
- It can't include the `static` modifier.
- The return type must be `void`.
- The declaration must include one parameter, which represents the right hand side of the compound assignment.

Beginning with C# 14, the increment (`++`) and decrement (`--`) operators can be overloaded as instance members. Instance operators can improve performance by avoiding the creation of a new instance. An instance operator must follow these rules:

- It must include the `public` modifier.
- It can't include the `static` modifier.
- The return type must be `void`.
- It can't declare any parameters, even if those parameters have a default value.

Non overloadable operators

The following table shows the operators that can't be overloaded:

Expand table

Operators	Alternatives
x && y, x || y	Overload both the true and false operators and the & or | operators. For more information, see User-defined conditional logical operators.
a[i], a?[i]	Define an indexer.
(T)x	Define custom type conversions performed by a cast expression. For more information, see User-defined conversion

Operators	Alternatives
	operators.
^x, x = y, x.y, x?.y, c ? t : f, x ?? y, ??= y, x..y, x->y, =>, f(x), as, await, checked, unchecked, default, delegate, is, nameof, new, sizeof, stackalloc, switch, typeof, with	None.

Before C# 14, the compound operators can't be overloaded. Overloading the corresponding binary operator implicitly overloads the corresponding compound assignment operator.

Operator overload resolution

> **Important**
>
> This section applies to C# 14 and later. Before C# 14, user-defined compound assignment operators and instance increment and decrement operators aren't allowed.

If `x` is classified as a variable in a compound assignment expression such as `x «op»= y`, instance operators are preferred over any static operator for `«op»`. If an overloaded `«op»=` operator isn't declared for the type of `x` or `x` isn't classified as a variable, the static operators are used.

For the postfix operator `++`, if `x` isn't classified as a variable *or* the expression `x++` is used, the instance `operator++` is ignored. Otherwise, preference is given to the instance `operator ++`. For example,

```
C#

x++; // Instance operator++ preferred.
y = x++; // instance operator++ isn't considered.
```

The reason for this rule is that `y` should be assigned to the value of `x` *before* it's incremented. The compiler can't determine that for a user-defined implementation in a reference type.

For the prefix operator `++`, if `x` is classified as a variable in `++x`, the instance operator is preferred over a static unary operator.

C# language specification

For more information, see the following sections of the C# language specification:

- Operator overloading
- Operators

See also

- C# operators and expressions
- User-defined conversion operators
- Design guidelines - Operator overloads
- Design guidelines - Equality operators
- Why are overloaded operators always static in C#?

Declaration statements

Article • 06/21/2023

A declaration statement declares a new local variable, local constant, or local reference variable. To declare a local variable, specify its type and provide its name. You can declare multiple variables of the same type in one statement, as the following example shows:

```
C#

string greeting;
int a, b, c;
List<double> xs;
```

In a declaration statement, you can also initialize a variable with its initial value:

```
C#

string greeting = "Hello";
int a = 3, b = 2, c = a + b;
List<double> xs = new();
```

The preceding examples explicitly specify the type of a variable. You can also let the compiler infer the type of a variable from its initialization expression. To do that, use the `var` keyword instead of a type's name. For more information, see the Implicitly-typed local variables section.

To declare a local constant, use the const keyword, as the following example shows:

```
C#

const string Greeting = "Hello";
const double MinLimit = -10.0, MaxLimit = -MinLimit;
```

When you declare a local constant, you must also initialize it.

For information about local reference variables, see the Reference variables section.

Implicitly-typed local variables

When you declare a local variable, you can let the compiler infer the type of the variable from the initialization expression. To do that use the `var` keyword instead of the name

of a type:

C#

```
var greeting = "Hello";
Console.WriteLine(greeting.GetType());  // output: System.String

var a = 32;
Console.WriteLine(a.GetType());  // output: System.Int32

var xs = new List<double>();
Console.WriteLine(xs.GetType());  // output:
System.Collections.Generic.List`1[System.Double]
```

As the preceding example shows, implicitly-typed local variables are strongly typed.

> **Note**
>
> When you use `var` in the enabled nullable aware context and the type of an initialization expression is a reference type, the compiler always infers a **nullable** reference type even if the type of an initialization expression isn't nullable.

A common use of `var` is with a constructor invocation expression. The use of `var` allows you to not repeat a type name in a variable declaration and object instantiation, as the following example shows:

C#

```
var xs = new List<int>();
```

You can use a target-typed new expression as an alternative:

C#

```
List<int> xs = new();
List<int>? ys = new();
```

When you work with anonymous types, you must use implicitly-typed local variables. The following example shows a query expression that uses an anonymous type to hold a customer's name and phone number:

C#

```
var fromPhoenix = from cust in customers
                  where cust.City == "Phoenix"
```

```
                    select new { cust.Name, cust.Phone };

foreach (var customer in fromPhoenix)
{
    Console.WriteLine($"Name={customer.Name}, Phone={customer.Phone}");
}
```

In the preceding example, you can't explicitly specify the type of the `fromPhoenix` variable. The type is IEnumerable<T> but in this case `T` is an anonymous type and you can't provide its name. That's why you need to use `var`. For the same reason, you must use `var` when you declare the `customer` iteration variable in the `foreach` statement.

For more information about implicitly-typed local variables, see Implicitly-typed local variables.

In pattern matching, the `var` keyword is used in a var pattern.

Reference variables

When you declare a local variable and add the `ref` keyword before the variable's type, you declare a *reference variable*, or a `ref` local:

C#

```
ref int aliasOfvariable = ref variable;
```

A reference variable is a variable that refers to another variable, which is called the *referent*. That is, a reference variable is an *alias* to its referent. When you assign a value to a reference variable, that value is assigned to the referent. When you read the value of a reference variable, the referent's value is returned. The following example demonstrates that behavior:

C#

```
int a = 1;
ref int aliasOfa = ref a;
Console.WriteLine($"(a, aliasOfa) is ({a}, {aliasOfa})");  // output: (a,
aliasOfa) is (1, 1)

a = 2;
Console.WriteLine($"(a, aliasOfa) is ({a}, {aliasOfa})");  // output: (a,
aliasOfa) is (2, 2)

aliasOfa = 3;
```

```
Console.WriteLine($"(a, aliasOfa) is ({a}, {aliasOfa})");  // output: (a,
aliasOfa) is (3, 3)
```

Use the ref assignment operator `= ref` to change the referent of a reference variable, as the following example shows:

C#

```
void Display(int[] s) => Console.WriteLine(string.Join(" ", s));

int[] xs = [0, 0, 0];
Display(xs);

ref int element = ref xs[0];
element = 1;
Display(xs);

element = ref xs[^1];
element = 3;
Display(xs);
// Output:
// 0 0 0
// 1 0 0
// 1 0 3
```

In the preceding example, the `element` reference variable is initialized as an alias to the first array element. Then it's `ref` reassigned to refer to the last array element.

You can define a `ref readonly` local variable. You can't assign a value to a `ref readonly` variable. However you can `ref` reassign such a reference variable, as the following example shows:

C#

```
int[] xs = [1, 2, 3];

ref readonly int element = ref xs[0];
// element = 100;  error CS0131: The left-hand side of an assignment must be
a variable, property or indexer
Console.WriteLine(element);  // output: 1

element = ref xs[^1];
Console.WriteLine(element);  // output: 3
```

You can assign a reference return to a reference variable, as the following example shows:

C#

```
using System;

public class NumberStore
{
    private readonly int[] numbers = [1, 30, 7, 1557, 381, 63, 1027, 2550,
511, 1023];

    public ref int GetReferenceToMax()
    {
        ref int max = ref numbers[0];
        for (int i = 1; i < numbers.Length; i++)
        {
            if (numbers[i] > max)
            {
                max = ref numbers[i];
            }
        }
        return ref max;
    }

    public override string ToString() => string.Join(" ", numbers);
}

public static class ReferenceReturnExample
{
    public static void Run()
    {
        var store = new NumberStore();
        Console.WriteLine($"Original sequence: {store.ToString()}");

        ref int max = ref store.GetReferenceToMax();
        max = 0;
        Console.WriteLine($"Updated sequence:  {store.ToString()}");
        // Output:
        // Original sequence: 1 30 7 1557 381 63 1027 2550 511 1023
        // Updated sequence:  1 30 7 1557 381 63 1027 0 511 1023
    }
}
```

In the preceding example, the `GetReferenceToMax` method is a *returns-by-ref* method. It doesn't return the maximum value itself, but a reference return that is an alias to the array element that holds the maximum value. The `Run` method assigns a reference return to the `max` reference variable. Then, by assigning to `max`, it updates the internal storage of the `store` instance. You can also define a `ref readonly` method. The callers of a `ref readonly` method can't assign a value to its reference return.

The iteration variable of the `foreach` statement can be a reference variable. For more information, see the foreach statement section of the Iteration statements article.

In performance-critical scenarios, the use of reference variables and returns might increase performance by avoiding potentially expensive copy operations.

The compiler ensures that a reference variable doesn't outlive its referent and stays valid for the whole of its lifetime. For more information, see the Ref safe contexts section of the C# language specification.

For information about the `ref` fields, see the ref fields section of the ref structure types article.

scoped ref

The contextual keyword `scoped` restricts the lifetime of a value. The `scoped` modifier restricts the *ref-safe-to-escape* or *safe-to-escape* lifetime, respectively, to the current method. Effectively, adding the `scoped` modifier asserts that your code won't extend the lifetime of the variable.

You can apply `scoped` to a parameter or local variable. The `scoped` modifier may be applied to parameters and locals when the type is a ref struct. Otherwise, the `scoped` modifier may be applied only to local reference variables. That includes local variables declared with the `ref` modifier and parameters declared with the `in`, `ref` or `out` modifiers.

The `scoped` modifier is implicitly added to `this` in methods declared in a `struct`, `out` parameters, and `ref` parameters when the type is a `ref struct`.

C# language specification

For more information, see the following sections of the C# language specification:

- Declaration statements
- Reference variables and returns

For more information about the `scoped` modifier, see the Low-level struct improvements proposal note.

See also

- Object and collection initializers
- ref keyword
- Reduce memory allocations using new C# features

- 'var' preferences (style rules IDE0007 and IDE0008)

Exception-handling statements - `throw`, `try-catch`, `try-finally`, and `try-catch-finally`

08/19/2025

You use the `throw` and `try` statements to work with exceptions. Use the throw statement to throw an exception. Use the try statement to catch and handle exceptions that might occur during execution of a code block.

The `throw` statement

The `throw` statement throws an exception:

C#

```
if (shapeAmount <= 0)
{
    throw new ArgumentOutOfRangeException(nameof(shapeAmount), "Amount of shapes
must be positive.");
}
```

In a `throw e;` statement, the result of expression `e` must be implicitly convertible to System.Exception.

You can use the built-in exception classes, for example, ArgumentOutOfRangeException or InvalidOperationException. .NET also provides the following helper methods to throw exceptions in certain conditions: ArgumentNullException.ThrowIfNull and ArgumentException.ThrowIfNullOrEmpty. You can also define your own exception classes that derive from System.Exception. For more information, see Creating and throwing exceptions.

Inside a catch block, you can use a `throw;` statement to re-throw the exception that is handled by the `catch` block:

C#

```
try
{
    ProcessShapes(shapeAmount);
}
catch (Exception e)
{
    LogError(e, "Shape processing failed.");
```

```
    throw;
}
```

> **Note**
>
> `throw;` preserves the original stack trace of the exception, which is stored in the Exception.StackTrace property. Opposite to that, `throw e;` updates the StackTrace property of `e`.

When an exception is thrown, the common language runtime (CLR) looks for the catch block that can handle this exception. If the currently executed method doesn't contain such a `catch` block, the CLR looks at the method that called the current method, and so on up the call stack. If no `catch` block is found, the CLR terminates the executing thread. For more information, see the How exceptions are handled section of the C# language specification.

The `throw` expression

You can also use `throw` as an expression. This might be convenient in a number of cases, which include:

- the conditional operator. The following example uses a `throw` expression to throw an ArgumentException when the passed array `args` is empty:

  C#

  ```
  string first = args.Length >= 1
      ? args[0]
      : throw new ArgumentException("Please supply at least one argument.");
  ```

- the null-coalescing operator. The following example uses a `throw` expression to throw an ArgumentNullException when the string to assign to a property is `null`:

  C#

  ```
  public string Name
  {
      get => name;
      set => name = value ??
          throw new ArgumentNullException(paramName: nameof(value), message:
  "Name cannot be null");
  }
  ```

- an expression-bodied lambda or method. The following example uses a `throw` expression to throw an InvalidCastException to indicate that a conversion to a DateTime value is not supported:

```
C#

DateTime ToDateTime(IFormatProvider provider) =>
         throw new InvalidCastException("Conversion to a DateTime is not
supported.");
```

The `try` statement

You can use the `try` statement in any of the following forms: try-catch - to handle exceptions that might occur during execution of the code inside a `try` block, try-finally - to specify the code that is executed when control leaves the `try` block, and try-catch-finally - as a combination of the preceding two forms.

The `try-catch` statement

Use the `try-catch` statement to handle exceptions that might occur during execution of a code block. Place the code where an exception might occur inside a `try` block. Use a *catch clause* to specify the base type of exceptions you want to handle in the corresponding `catch` block:

```
C#

try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

You can provide several catch clauses:

```
C#

try
{
    var result = await ProcessAsync(-3, 4, cancellationToken);
    Console.WriteLine($"Processing succeeded: {result}");
```

```
}
catch (ArgumentException e)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
catch (OperationCanceledException)
{
    Console.WriteLine("Processing is cancelled.");
}
```

When an exception occurs, catch clauses are examined in the specified order, from top to bottom. At maximum, only one `catch` block is executed for any thrown exception. As the preceding example also shows, you can omit declaration of an exception variable and specify only the exception type in a catch clause. A catch clause without any specified exception type matches any exception and, if present, must be the last catch clause.

If you want to re-throw a caught exception, use the throw statement, as the following example shows:

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e)
{
    LogError(e, "Processing failed.");
    throw;
}
```

> **Note**
>
> `throw;` preserves the original stack trace of the exception, which is stored in the Exception.StackTrace property. Opposite to that, `throw e;` updates the StackTrace property of `e`.

A `when` exception filter

Along with an exception type, you can also specify an exception filter that further examines an exception and decides if the corresponding `catch` block handles that exception. An exception filter is a Boolean expression that follows the `when` keyword, as the following example shows:

```
```

C#

```
try
{
    var result = Process(-3, 4);
    Console.WriteLine($"Processing succeeded: {result}");
}
catch (Exception e) when (e is ArgumentException || e is DivideByZeroException)
{
    Console.WriteLine($"Processing failed: {e.Message}");
}
```

The preceding example uses an exception filter to provide a single `catch` block to handle exceptions of two specified types.

You can provide several `catch` clauses for the same exception type if they distinguish by exception filters. One of those clauses might have no exception filter. If such a clause exists, it must be the last of the clauses that specify that exception type.

If a `catch` clause has an exception filter, it can specify the exception type that is the same as or less derived than an exception type of a `catch` clause that appears after it. For example, if an exception filter is present, a `catch (Exception e)` clause doesn't need to be the last clause.

Exception filters vs. traditional exception handling

Exception filters provide significant advantages over traditional exception handling approaches. The key difference is **when** the exception handling logic is evaluated:

- **Exception filters (`when`)**: The filter expression is evaluated *before* the stack is unwound. This means the original call stack and all local variables remain intact during filter evaluation.
- **Traditional `catch` blocks**: The catch block executes *after* the stack is unwound, potentially losing valuable debugging information.

Here's a comparison showing the difference:

C#

```
public static void DemonstrateStackUnwindingDifference()
{
    var localVariable = "Important debugging info";

    try
    {
        ProcessWithExceptionFilter(localVariable);
    }
    catch (InvalidOperationException ex) when (ex.Message.Contains("filter"))
```

```
    {
        // Exception filter: Stack not unwound yet.
        // localVariable is still accessible in debugger.
        // Call stack shows original throwing location.
        Console.WriteLine($"Caught with filter: {ex.Message}");
        Console.WriteLine($"Local variable accessible: {localVariable}");
    }

    try
    {
        ProcessWithTraditionalCatch(localVariable);
    }
    catch (InvalidOperationException ex)
    {
        // Traditional catch: Stack already unwound.
        // Some debugging information may be lost.
        if (ex.Message.Contains("traditional"))
        {
            Console.WriteLine($"Caught with if: {ex.Message}");
            Console.WriteLine($"Local variable accessible: {localVariable}");
        }
        else
        {
            throw; // Re-throws and further modifies stack trace.
        }
    }
}

private static void ProcessWithExceptionFilter(string context)
{
    throw new InvalidOperationException($"Exception for filter demo: {context}");
}

private static void ProcessWithTraditionalCatch(string context)
{
    throw new InvalidOperationException($"Exception for traditional demo:
{context}");
}
```

Advantages of exception filters

- **Better debugging experience**: Since the stack isn't unwound until a filter matches, debuggers can show the original point of failure with all local variables intact.
- **Performance benefits**: If no filter matches, the exception continues propagating without the overhead of stack unwinding and restoration.
- **Cleaner code**: Multiple filters can handle different conditions of the same exception type without requiring nested if-else statements.
- **Logging and diagnostics**: You can examine and log exception details before deciding whether to handle the exception:

C#

```
public static void DemonstrateDebuggingAdvantage()
{
    var contextData = new Dictionary<string, object>
    {
        ["RequestId"] = Guid.NewGuid(),
        ["UserId"] = "user123",
        ["Timestamp"] = DateTime.Now
    };

    try
    {
        // Simulate a deep call stack.
        Level1Method(contextData);
    }
    catch (Exception ex) when (LogAndFilter(ex, contextData))
    {
        // This catch block may never execute if LogAndFilter returns false.
        // But LogAndFilter can examine the exception while the stack is intact.
        Console.WriteLine("Exception handled after logging");
    }
}

private static void Level1Method(Dictionary<string, object> context)
{
    Level2Method(context);
}

private static void Level2Method(Dictionary<string, object> context)
{
    Level3Method(context);
}

private static void Level3Method(Dictionary<string, object> context)
{
    throw new InvalidOperationException("Error in deep call stack");
}

private static bool LogAndFilter(Exception ex, Dictionary<string, object> context)
{
    // This method runs before stack unwinding.
    // Full call stack and local variables are still available.
    Console.WriteLine($"Exception occurred: {ex.Message}");
    Console.WriteLine($"Request ID: {context["RequestId"]}");
    Console.WriteLine($"Full stack trace preserved: {ex.StackTrace}");

    // Return true to handle the exception, false to continue search.
    return ex.Message.Contains("deep call stack");
}
```

When to use exception filters

Use exception filters when you need to:

- Handle exceptions based on specific conditions or properties.
- Preserve the original call stack for debugging.
- Log or examine exceptions before deciding whether to handle them.
- Handle the same exception type differently based on context.

C#

```
public static void HandleFileOperations(string filePath)
{
    try
    {
        // Simulate file operation that might fail.
        ProcessFile(filePath);
    }
    catch (IOException ex) when (ex.Message.Contains("access denied"))
    {
        Console.WriteLine("File access denied. Check permissions.");
    }
    catch (IOException ex) when (ex.Message.Contains("not found"))
    {
        Console.WriteLine("File not found. Verify the path.");
    }
    catch (IOException ex) when (IsNetworkPath(filePath))
    {
        Console.WriteLine($"Network file operation failed: {ex.Message}");
    }
    catch (IOException)
    {
        Console.WriteLine("Other I/O error occurred.");
    }
}

private static void ProcessFile(string filePath)
{
    // Simulate different types of file exceptions.
    if (filePath.Contains("denied"))
        throw new IOException("File access denied");
    if (filePath.Contains("missing"))
        throw new IOException("File not found");
    if (IsNetworkPath(filePath))
        throw new IOException("Network timeout occurred");
}

private static bool IsNetworkPath(string path)
{
    return path.StartsWith(@"\\") || path.StartsWith("http");
}
```

Stack trace preservation

Exception filters preserve the original `ex.StackTrace` property. If a `catch` clause can't process the exception and re-throws, the original stack information is lost. The `when` filter doesn't unwind the stack, so if a `when` filter is `false`, the original stack trace isn't changed.

The exception filter approach is valuable in applications where preserving debugging information is crucial for diagnosing issues.

Exceptions in async and iterator methods

If an exception occurs in an async function, it propagates to the caller of the function when you await the result of the function, as the following example shows:

C#

```
public static async Task Run()
{
    try
    {
        Task<int> processing = ProcessAsync(-1);
        Console.WriteLine("Launched processing.");

        int result = await processing;
        Console.WriteLine($"Result: {result}.");
    }
    catch (ArgumentException e)
    {
        Console.WriteLine($"Processing failed: {e.Message}");
    }
    // Output:
    // Launched processing.
    // Processing failed: Input must be non-negative. (Parameter 'input')
}

private static async Task<int> ProcessAsync(int input)
{
    if (input < 0)
    {
        throw new ArgumentOutOfRangeException(nameof(input), "Input must be non-
negative.");
    }

    await Task.Delay(500);
    return input;
}
```

If an exception occurs in an iterator method, it propagates to the caller only when the iterator advances to the next element.

The `try-finally` statement

In a `try-finally` statement, the `finally` block is executed when control leaves the `try` block. Control might leave the `try` block as a result of

- normal execution,
- execution of a jump statement (that is, `return`, `break`, `continue`, or `goto`), or
- propagation of an exception out of the `try` block.

The following example uses the `finally` block to reset the state of an object before control leaves the method:

C#

```
public async Task HandleRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    finally
    {
        Busy = false;
    }
}
```

You can also use the `finally` block to clean up allocated resources used in the `try` block.

> **Note**
>
> When the type of a resource implements the IDisposable or IAsyncDisposable interface, consider the using statement. The `using` statement ensures that acquired resources are disposed when control leaves the `using` statement. The compiler transforms a `using` statement into a `try-finally` statement.

Execution of the `finally` block depends on whether the operating system chooses to trigger an exception unwind operation. The only cases where `finally` blocks aren't executed involve immediate termination of a program. For example, such a termination might happen because of the Environment.FailFast call or an OverflowException or InvalidProgramException exception. Most operating systems perform a reasonable resource clean-up as part of stopping and unloading the process.

The `try-catch-finally` statement

You use a `try-catch-finally` statement both to handle exceptions that might occur during execution of the `try` block and specify the code that must be executed when control leaves the `try` statement:

```
C#
```

```
public async Task ProcessRequest(int itemId, CancellationToken ct)
{
    Busy = true;

    try
    {
        await ProcessAsync(itemId, ct);
    }
    catch (Exception e) when (e is not OperationCanceledException)
    {
        LogError(e, $"Failed to process request for item ID {itemId}.");
        throw;
    }
    finally
    {
        Busy = false;
    }

}
```

When an exception is handled by a `catch` block, the `finally` block is executed after execution of that `catch` block (even if another exception occurs during execution of the `catch` block). For information about `catch` and `finally` blocks, see The try-catch statement and The try-finally statement sections, respectively.

C# language specification

For more information, see the following sections of the C# language specification:

- The throw statement
- The try statement
- Exceptions

See also

- Exceptions and exception handling
- Handling and throwing exceptions in .NET

- throw preferences (style rule IDE0016)
- AppDomain.FirstChanceException
- AppDomain.UnhandledException
- TaskScheduler.UnobservedTaskException

Iteration statements - `for`, `foreach`, `do`, and `while`

Article • 11/14/2023

The iteration statements repeatedly execute a statement or a block of statements. The for statement executes its body while a specified Boolean expression evaluates to `true`. The foreach statement enumerates the elements of a collection and executes its body for each element of the collection. The do statement conditionally executes its body one or more times. The while statement conditionally executes its body zero or more times.

At any point within the body of an iteration statement, you can break out of the loop using the break statement. You can step to the next iteration in the loop using the continue statement.

The `for` statement

The `for` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. The following example shows the `for` statement that executes its body while an integer counter is less than three:

```
C#
```

```
for (int i = 0; i < 3; i++)
{
    Console.Write(i);
}
// Output:
// 012
```

The preceding example shows the elements of the `for` statement:

- The *initializer* section that is executed only once, before entering the loop. Typically, you declare and initialize a local loop variable in that section. The declared variable can't be accessed from outside the `for` statement.

  The *initializer* section in the preceding example declares and initializes an integer counter variable:

  ```
  C#
  ```

  ```
  int i = 0
  ```

- The *condition* section that determines if the next iteration in the loop should be executed. If it evaluates to `true` or isn't present, the next iteration is executed; otherwise, the loop is exited. The *condition* section must be a Boolean expression.

  The *condition* section in the preceding example checks if a counter value is less than three:

  ```
  C#

  i < 3
  ```

- The *iterator* section that defines what happens after each execution of the body of the loop.

  The *iterator* section in the preceding example increments the counter:

  ```
  C#

  i++
  ```

- The body of the loop, which must be a statement or a block of statements.

The iterator section can contain zero or more of the following statement expressions, separated by commas:

- prefix or postfix increment expression, such as `++i` or `i++`
- prefix or postfix decrement expression, such as `--i` or `i--`
- assignment
- invocation of a method
- await expression
- creation of an object by using the new operator

If you don't declare a loop variable in the initializer section, you can use zero or more of the expressions from the preceding list in the initializer section as well. The following example shows several less common usages of the initializer and iterator sections: assigning a value to an external variable in the initializer section, invoking a method in both the initializer and the iterator sections, and changing the values of two variables in the iterator section:

```
C#

int i;
int j = 3;
for (i = 0, Console.WriteLine($"Start: i={i}, j={j}"); i < j; i++, j--,
```

```
Console.WriteLine($"Step: i={i}, j={j}"))
{
    //...
}
// Output:
// Start: i=0, j=3
// Step: i=1, j=2
// Step: i=2, j=1
```

All the sections of the `for` statement are optional. For example, the following code defines the infinite `for` loop:

C#

```
for ( ; ; )
{
    //...
}
```

The `foreach` statement

The `foreach` statement executes a statement or a block of statements for each element in an instance of the type that implements the System.Collections.IEnumerable or System.Collections.Generic.IEnumerable<T> interface, as the following example shows:

C#

```
List<int> fibNumbers = new() { 0, 1, 1, 2, 3, 5, 8, 13 };
foreach (int element in fibNumbers)
{
    Console.Write($"{element} ");
}
// Output:
// 0 1 1 2 3 5 8 13
```

The `foreach` statement isn't limited to those types. You can use it with an instance of any type that satisfies the following conditions:

- A type has the public parameterless `GetEnumerator` method. The `GetEnumerator` method can be a type's extension method.
- The return type of the `GetEnumerator` method has the public `Current` property and the public parameterless `MoveNext` method whose return type is `bool`.

The following example uses the `foreach` statement with an instance of the System.Span<T> type, which doesn't implement any interfaces:

C#

```
Span<int> numbers = [3, 14, 15, 92, 6];
foreach (int number in numbers)
{
    Console.Write($"{number} ");
}
// Output:
// 3 14 15 92 6
```

If the enumerator's `Current` property returns a reference return value (`ref T` where `T` is the type of a collection element), you can declare an iteration variable with the `ref` or `ref readonly` modifier, as the following example shows:

C#

```
Span<int> storage = stackalloc int[10];
int num = 0;
foreach (ref int item in storage)
{
    item = num++;
}
foreach (ref readonly var item in storage)
{
    Console.Write($"{item} ");
}
// Output:
// 0 1 2 3 4 5 6 7 8 9
```

If the source collection of the `foreach` statement is empty, the body of the `foreach` statement isn't executed and skipped. If the `foreach` statement is applied to `null`, a NullReferenceException is thrown.

await foreach

You can use the `await foreach` statement to consume an asynchronous stream of data, that is, the collection type that implements the IAsyncEnumerable<T> interface. Each iteration of the loop may be suspended while the next element is retrieved asynchronously. The following example shows how to use the `await foreach` statement:

C#

```
await foreach (var item in GenerateSequenceAsync())
{
```

```
    Console.WriteLine(item);
}
```

You can also use the `await foreach` statement with an instance of any type that satisfies the following conditions:

- A type has the public parameterless `GetAsyncEnumerator` method. That method can be a type's extension method.
- The return type of the `GetAsyncEnumerator` method has the public `Current` property and the public parameterless `MoveNextAsync` method whose return type is Task<bool>, ValueTask<bool>, or any other awaitable type whose awaiter's `GetResult` method returns a `bool` value.

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 Consuming the Task-based asynchronous pattern. For more information about asynchronous streams, see the Asynchronous streams tutorial.

Type of an iteration variable

You can use the var keyword to let the compiler infer the type of an iteration variable in the `foreach` statement, as the following code shows:

C#

```
foreach (var item in collection) { }
```

> **Note**
>
> Type of `var` can be inferred by the compiler as a nullable reference type, depending on whether the nullable aware context is enabled and whether the type of an initialization expression is a reference type. For more information see Implicitly-typed local variables.

You can also explicitly specify the type of an iteration variable, as the following code shows:

C#

```
IEnumerable<T> collection = new T[5];
foreach (V item in collection) { }
```

In the preceding form, type `T` of a collection element must be implicitly or explicitly convertible to type `V` of an iteration variable. If an explicit conversion from `T` to `V` fails at run time, the `foreach` statement throws an InvalidCastException. For example, if `T` is a non-sealed class type, `V` can be any interface type, even the one that `T` doesn't implement. At run time, the type of a collection element may be the one that derives from `T` and actually implements `V`. If that's not the case, an InvalidCastException is thrown.

The `do` statement

The `do` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. Because that expression is evaluated after each execution of the loop, a `do` loop executes one or more times. The `do` loop differs from the while loop, which executes zero or more times.

The following example shows the usage of the `do` statement:

C#

```
int n = 0;
do
{
    Console.Write(n);
    n++;
} while (n < 5);
// Output:
// 01234
```

The `while` statement

The `while` statement executes a statement or a block of statements while a specified Boolean expression evaluates to `true`. Because that expression is evaluated before each execution of the loop, a `while` loop executes zero or more times. The `while` loop differs from the do loop, which executes one or more times.

The following example shows the usage of the `while` statement:

C#

```
int n = 0;
while (n < 5)
{
    Console.Write(n);
    n++;
}
// Output:
// 01234
```

C# language specification

For more information, see the following sections of the C# language specification:

- The for statement
- The foreach statement
- The do statement
- The while statement

For more information about these features, see the following feature proposal notes:

- Async streams
- Extension GetEnumerator support for foreach loops

See also

- Declarations
- Iterators

Selection statements - `if`, `if-else`, and `switch`

Article • 02/21/2025

The `if`, `if-else` and `switch` statements select statements to execute from many possible paths based on the value of an expression. The if statement executes a statement only if a provided Boolean expression evaluates to `true`. The if-else statement allows you to choose which of the two code paths to follow based on a Boolean expression. The switch statement selects a statement list to execute based on a pattern match with an expression.

The `if` statement

An `if` statement can be any of the following two forms:

- An `if` statement with an `else` part selects one of the two statements to execute based on the value of a Boolean expression, as the following example shows:

  ```
  C#

  DisplayWeatherReport(15.0);  // Output: Cold.
  DisplayWeatherReport(24.0);  // Output: Perfect!

  void DisplayWeatherReport(double tempInCelsius)
  {
      if (tempInCelsius < 20.0)
      {
          Console.WriteLine("Cold.");
      }
      else
      {
          Console.WriteLine("Perfect!");
      }
  }
  ```

- An `if` statement without an `else` part executes its body only if a Boolean expression evaluates to `true`, as the following example shows:

  ```
  C#

  DisplayMeasurement(45);  // Output: The measurement value is 45
  DisplayMeasurement(-3);  // Output: Warning: not acceptable value! The
  measurement value is -3
  ```

```
void DisplayMeasurement(double value)
{
    if (value < 0 || value > 100)
    {
        Console.Write("Warning: not acceptable value! ");
    }

    Console.WriteLine($"The measurement value is {value}");
}
```

You can nest `if` statements to check multiple conditions, as the following example shows:

C#

```
DisplayCharacter('f');  // Output: A lowercase letter: f
DisplayCharacter('R');  // Output: An uppercase letter: R
DisplayCharacter('8');  // Output: A digit: 8
DisplayCharacter(',');  // Output: Not alphanumeric character: ,

void DisplayCharacter(char ch)
{
    if (char.IsUpper(ch))
    {
        Console.WriteLine($"An uppercase letter: {ch}");
    }
    else if (char.IsLower(ch))
    {
        Console.WriteLine($"A lowercase letter: {ch}");
    }
    else if (char.IsDigit(ch))
    {
        Console.WriteLine($"A digit: {ch}");
    }
    else
    {
        Console.WriteLine($"Not alphanumeric character: {ch}");
    }
}
```

In an expression context, you can use the conditional operator ?: to evaluate one of the two expressions based on the value of a Boolean expression.

The `switch` statement

The `switch` statement selects a statement list to execute based on a pattern match with a match expression, as the following example shows:

C#

```
DisplayMeasurement(-4);  // Output: Measured value is -4; too low.
DisplayMeasurement(5);  // Output: Measured value is 5.
DisplayMeasurement(30);  // Output: Measured value is 30; too high.
DisplayMeasurement(double.NaN);  // Output: Failed measurement.

void DisplayMeasurement(double measurement)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too
high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

At the preceding example, the `switch` statement uses the following patterns:

- A relational pattern: to compare an expression result with a constant.
- A constant pattern: test if an expression result equals a constant.

> **ⓘ Important**
>
> For information about the patterns supported by the `switch` statement, see Patterns.

The preceding example also demonstrates the `default` case. The `default` case specifies statements to execute when a match expression doesn't match any other case pattern. If a match expression doesn't match any case pattern and there's no `default` case, control falls through a `switch` statement.

A `switch` statement executes the *statement list* in the first *switch section* whose *case pattern* matches a match expression and whose case guard, if present, evaluates to `true`. A `switch` statement evaluates case patterns in text order from top to bottom. The

compiler generates an error when a `switch` statement contains an unreachable case. That is a case that is already handled by an upper case or whose pattern is impossible to match.

> **Note**
>
> The `default` case can appear in any place within a `switch` statement. Regardless of its position, the `default` case is evaluated only if all other case patterns aren't matched or the `goto default;` statement is executed in one of the switch sections.

You can specify multiple case patterns for one section of a `switch` statement, as the following example shows:

C#

```
DisplayMeasurement(-4);  // Output: Measured value is -4; out of an
acceptable range.
DisplayMeasurement(50);  // Output: Measured value is 50.
DisplayMeasurement(132);  // Output: Measured value is 132; out of an
acceptable range.

void DisplayMeasurement(int measurement)
{
    switch (measurement)
    {
        case < 0:
        case > 100:
            Console.WriteLine($"Measured value is {measurement}; out of an
acceptable range.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
```

Within a `switch` statement, control can't fall through from one switch section to the next. As the examples in this section show, typically you use the `break` statement at the end of each switch section to pass control out of a `switch` statement. You can also use the return and throw statements to pass control out of a `switch` statement. To imitate the fall-through behavior and pass control to other switch section, you can use the goto statement.

> **Important**

Every *switch section* must end with a `break`, `goto` or `return`. Falling through from one switch section to the next generates a compiler error. However, multiple switch labels can be applied to the same switch section, like `case < 0:` in example above. This deliberate design choice allows for concisely handling multiple cases that share the same or interdependent logic.

In an expression context, you can use the switch expression to evaluate a single expression from a list of candidate expressions based on a pattern match with an expression.

> **ⓘ Important**
>
> Differences between **switch expression** and **switch statement**:
>
> - **switch statement** is used to control the execution flow within a block of code.
> - **switch expression** is typically used in contexts of value return and value assignment, often as a expression-bodied members.
> - a **switch expression** case section cannot be empty, a **switch statement** can.

Case guards

A case pattern may not be expressive enough to specify the condition for the execution of the switch section. In such a case, you can use a *case guard*. That is an additional condition that must be satisfied together with a matched pattern. A case guard must be a Boolean expression. You specify a case guard after the `when` keyword that follows a pattern, as the following example shows:

C#

```
DisplayMeasurements(3, 4);  // Output: First measurement is 3, second
measurement is 4.
DisplayMeasurements(5, 5);  // Output: Both measurements are valid and equal
to 5.

void DisplayMeasurements(int a, int b)
{
    switch ((a, b))
    {
        case (> 0, > 0) when a == b:
            Console.WriteLine($"Both measurements are valid and equal to
{a}.");
            break;

        case (> 0, > 0):
```

```
            Console.WriteLine($"First measurement is {a}, second measurement
is {b}.");
            break;

        default:
            Console.WriteLine("One or both measurements are not valid.");
            break;
    }
}
```

The preceding example uses positional patterns with nested relational patterns.

C# language specification

For more information, see the following sections of the C# language specification:

- The if statement
- The switch statement

For more information about patterns, see the Patterns and pattern matching section of the C# language specification.

See also

- Conditional operator ?:
- Logical operators
- Patterns
- switch expression
- Add missing cases to switch statement (style rule IDE0010)

Jump statements - `break`, `continue`, `return`, and `goto`

Article • 03/15/2023

The jump statements unconditionally transfer control. The break statement terminates the closest enclosing iteration statement or switch statement. The continue statement starts a new iteration of the closest enclosing iteration statement. The return statement terminates execution of the function in which it appears and returns control to the caller. The goto statement transfers control to a statement that is marked by a label.

For information about the `throw` statement that throws an exception and unconditionally transfers control as well, see The throw statement section of the Exception-handling statements article.

The `break` statement

The `break` statement terminates the closest enclosing iteration statement (that is, `for`, `foreach`, `while`, or `do` loop) or switch statement. The `break` statement transfers control to the statement that follows the terminated statement, if any.

C#

```
int[] numbers = [0, 1, 2, 3, 4, 5, 6, 7, 8, 9];
foreach (int number in numbers)
{
    if (number == 3)
    {
        break;
    }

    Console.Write($"{number} ");
}
Console.WriteLine();
Console.WriteLine("End of the example.");
// Output:
// 0 1 2
// End of the example.
```

In nested loops, the `break` statement terminates only the innermost loop that contains it, as the following example shows:

C#

```
for (int outer = 0; outer < 5; outer++)
{
    for (int inner = 0; inner < 5; inner++)
    {
        if (inner > outer)
        {
            break;
        }

        Console.Write($"{inner} ");
    }
    Console.WriteLine();
}
// Output:
// 0
// 0 1
// 0 1 2
// 0 1 2 3
// 0 1 2 3 4
```

When you use the `switch` statement inside a loop, a `break` statement at the end of a switch section transfers control only out of the `switch` statement. The loop that contains the `switch` statement is unaffected, as the following example shows:

C#

```
double[] measurements = [-4, 5, 30, double.NaN];
foreach (double measurement in measurements)
{
    switch (measurement)
    {
        case < 0.0:
            Console.WriteLine($"Measured value is {measurement}; too low.");
            break;

        case > 15.0:
            Console.WriteLine($"Measured value is {measurement}; too
high.");
            break;

        case double.NaN:
            Console.WriteLine("Failed measurement.");
            break;

        default:
            Console.WriteLine($"Measured value is {measurement}.");
            break;
    }
}
// Output:
// Measured value is -4; too low.
```

```
// Measured value is 5.
// Measured value is 30; too high.
// Failed measurement.
```

The `continue` statement

The `continue` statement starts a new iteration of the closest enclosing iteration statement (that is, `for`, `foreach`, `while`, or `do` loop), as the following example shows:

C#

```
for (int i = 0; i < 5; i++)
{
    Console.Write($"Iteration {i}: ");

    if (i < 3)
    {
        Console.WriteLine("skip");
        continue;
    }

    Console.WriteLine("done");
}
// Output:
// Iteration 0: skip
// Iteration 1: skip
// Iteration 2: skip
// Iteration 3: done
// Iteration 4: done
```

The `return` statement

The `return` statement terminates execution of the function in which it appears and returns control and the function's result, if any, to the caller.

If a function member doesn't compute a value, you use the `return` statement without expression, as the following example shows:

C#

```
Console.WriteLine("First call:");
DisplayIfNecessary(6);

Console.WriteLine("Second call:");
DisplayIfNecessary(5);

void DisplayIfNecessary(int number)
```

```
{
    if (number % 2 == 0)
    {
        return;
    }

    Console.WriteLine(number);
}
// Output:
// First call:
// Second call:
// 5
```

As the preceding example shows, you typically use the `return` statement without expression to terminate a function member early. If a function member doesn't contain the `return` statement, it terminates after its last statement is executed.

If a function member computes a value, you use the `return` statement with an expression, as the following example shows:

C#

```
double surfaceArea = CalculateCylinderSurfaceArea(1, 1);
Console.WriteLine($"{surfaceArea:F2}"); // output: 12.57

double CalculateCylinderSurfaceArea(double baseRadius, double height)
{
    double baseArea = Math.PI * baseRadius * baseRadius;
    double sideArea = 2 * Math.PI * baseRadius * height;
    return 2 * baseArea + sideArea;
}
```

When the `return` statement has an expression, that expression must be implicitly convertible to the return type of a function member unless it's async. The expression returned from an `async` function must be implicitly convertible to the type argument of Task<TResult> or ValueTask<TResult>, whichever is the return type of the function. If the return type of an `async` function is Task or ValueTask, you use the `return` statement without expression.

Ref returns

By default, the `return` statement returns the value of an expression. You can return a reference to a variable. Reference return values (or ref returns) are values that a method returns by reference to the caller. That is, the caller can modify the value returned by a method, and that change is reflected in the state of the object in the called method. To

do that, use the `return` statement with the `ref` keyword, as the following example shows:

C#

```
int[] xs = new int [] {10, 20, 30, 40 };
ref int found = ref FindFirst(xs, s => s == 30);
found = 0;
Console.WriteLine(string.Join(" ", xs));  // output: 10 20 0 40

ref int FindFirst(int[] numbers, Func<int, bool> predicate)
{
    for (int i = 0; i < numbers.Length; i++)
    {
        if (predicate(numbers[i]))
        {
            return ref numbers[i];
        }
    }
    throw new InvalidOperationException("No element satisfies the given
condition.");
}
```

A reference return value allows a method to return a reference to a variable, rather than a value, back to a caller. The caller can then choose to treat the returned variable as if it were returned by value or by reference. The caller can create a new variable that is itself a reference to the returned value, called a ref local. A *reference return value* means that a method returns a *reference* (or an alias) to some variable. That variable's scope must include the method. That variable's lifetime must extend beyond the return of the method. Modifications to the method's return value by the caller are made to the variable that is returned by the method.

Declaring that a method returns a *reference return value* indicates that the method returns an alias to a variable. The design intent is often that calling code accesses that variable through the alias, including to modify it. Methods returning by reference can't have the return type `void`.

In order for the caller to modify the object's state, the reference return value must be stored to a variable that is explicitly defined as a reference variable.

The `ref` return value is an alias to another variable in the called method's scope. You can interpret any use of the ref return as using the variable it aliases:

- When you assign its value, you're assigning a value to the variable it aliases.
- When you read its value, you're reading the value of the variable it aliases.
- If you return it *by reference*, you're returning an alias to that same variable.

- If you pass it to another method *by reference*, you're passing a reference to the variable it aliases.
- When you make a ref local alias, you make a new alias to the same variable.

A ref return must be *ref-safe-context* to the calling method. That means:

- The return value must have a lifetime that extends beyond the execution of the method. In other words, it can't be a local variable in the method that returns it. It can be an instance or static field of a class, or it can be an argument passed to the method. Attempting to return a local variable generates compiler error CS8168, "Can't return local 'obj' by reference because it isn't a ref local."
- The return value can't be the literal `null`. A method with a ref return can return an alias to a variable whose value is currently the `null` (uninstantiated) value or a nullable value type for a value type.
- The return value can't be a constant, an enumeration member, the by-value return value from a property, or a method of a `class` or `struct`.

In addition, reference return values aren't allowed on async methods. An asynchronous method may return before it has finished execution, while its return value is still unknown.

A method that returns a *reference return value* must:

- Include the ref keyword in front of the return type.
- Each return statement in the method body includes the ref keyword in front of the name of the returned instance.

The following example shows a method that satisfies those conditions and returns a reference to a `Person` object named `p`:

C#

```
public ref Person GetContactInformation(string fname, string lname)
{
    // ...method implementation...
    return ref p;
}
```

Here's a more complete ref return example, showing both the method signature and method body.

C#

```
public static ref int Find(int[,] matrix, Func<int, bool> predicate)
{
```

```
    for (int i = 0; i < matrix.GetLength(0); i++)
        for (int j = 0; j < matrix.GetLength(1); j++)
            if (predicate(matrix[i, j]))
                return ref matrix[i, j];
    throw new InvalidOperationException("Not found");
}
```

The called method may also declare the return value as `ref readonly` to return the value by reference, and enforce that the calling code can't modify the returned value. The calling method can avoid copying the returned value by storing the value in a local `ref readonly` reference variable.

The following example defines a `Book` class that has two String fields, `Title` and `Author`. It also defines a `BookCollection` class that includes a private array of `Book` objects. Individual book objects are returned by reference by calling its `GetBookByTitle` method.

C#

```
public class Book
{
    public string Author;
    public string Title;
}

public class BookCollection
{
    private Book[] books = { new Book { Title = "Call of the Wild, The",
Author = "Jack London" },
                             new Book { Title = "Tale of Two Cities, A", Author =
"Charles Dickens" }
                           };
    private Book nobook = null;

    public ref Book GetBookByTitle(string title)
    {
        for (int ctr = 0; ctr < books.Length; ctr++)
        {
            if (title == books[ctr].Title)
                return ref books[ctr];
        }
        return ref nobook;
    }

    public void ListBooks()
    {
        foreach (var book in books)
        {
            Console.WriteLine($"{book.Title}, by {book.Author}");
        }
        Console.WriteLine();
```

```
    }
}
```

When the caller stores the value returned by the `GetBookByTitle` method as a ref local, changes that the caller makes to the return value are reflected in the `BookCollection` object, as the following example shows.

C#

```
var bc = new BookCollection();
bc.ListBooks();

ref var book = ref bc.GetBookByTitle("Call of the Wild, The");
if (book != null)
    book = new Book { Title = "Republic, The", Author = "Plato" };
bc.ListBooks();
// The example displays the following output:
//       Call of the Wild, The, by Jack London
//       Tale of Two Cities, A, by Charles Dickens
//
//       Republic, The, by Plato
//       Tale of Two Cities, A, by Charles Dickens
```

The `goto` statement

The `goto` statement transfers control to a statement that is marked by a label, as the following example shows:

C#

```
var matrices = new Dictionary<string, int[][]>
{
    ["A"] =
    [
        [1, 2, 3, 4],
        [4, 3, 2, 1]
    ],
    ["B"] =
    [
        [5, 6, 7, 8],
        [8, 7, 6, 5]
    ],
};

CheckMatrices(matrices, 4);

void CheckMatrices(Dictionary<string, int[][]> matrixLookup, int target)
{
    foreach (var (key, matrix) in matrixLookup)
```

```
    {
        for (int row = 0; row < matrix.Length; row++)
        {
            for (int col = 0; col < matrix[row].Length; col++)
            {
                if (matrix[row][col] == target)
                {
                    goto Found;
                }
            }
        }
        Console.WriteLine($"Not found {target} in matrix {key}.");
        continue;

    Found:
        Console.WriteLine($"Found {target} in matrix {key}.");
    }
}
// Output:
// Found 4 in matrix A.
// Not found 4 in matrix B.
```

As the preceding example shows, you can use the `goto` statement to get out of a nested loop.

> **Tip**
>
> When you work with nested loops, consider refactoring separate loops into separate methods. That may lead to a simpler, more readable code without the `goto` statement.

You can also use the `goto` statement in the switch statement to transfer control to a switch section with a constant case label, as the following example shows:

C#

```
using System;

public enum CoffeeChoice
{
    Plain,
    WithMilk,
    WithIceCream,
}

public class GotoInSwitchExample
{
    public static void Main()
    {
```

```
        Console.WriteLine(CalculatePrice(CoffeeChoice.Plain));  // output:
10.0
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithMilk));  //
output: 15.0
        Console.WriteLine(CalculatePrice(CoffeeChoice.WithIceCream));  //
output: 17.0
    }

    private static decimal CalculatePrice(CoffeeChoice choice)
    {
        decimal price = 0;
        switch (choice)
        {
            case CoffeeChoice.Plain:
                price += 10.0m;
                break;

            case CoffeeChoice.WithMilk:
                price += 5.0m;
                goto case CoffeeChoice.Plain;

            case CoffeeChoice.WithIceCream:
                price += 7.0m;
                goto case CoffeeChoice.Plain;
        }
        return price;
    }
}
```

Within the `switch` statement, you can also use the statement `goto default;` to transfer control to the switch section with the `default` label.

If a label with the given name doesn't exist in the current function member, or if the `goto` statement isn't within the scope of the label, a compile-time error occurs. That is, you can't use the `goto` statement to transfer control out of the current function member or into any nested scope.

C# language specification

For more information, see the following sections of the C# language specification:

- The break statement
- The continue statement
- The return statement
- The goto statement

See also

- yield statement

The checked and unchecked statements (C# reference)

Article • 11/01/2024

The `checked` and `unchecked` statements specify the overflow-checking context for integral-type arithmetic operations and conversions. The default statement is `unchecked`. When integer arithmetic overflow occurs, the overflow-checking context defines what happens. In a checked context, a System.OverflowException is thrown; if overflow happens in a constant expression, a compile-time error occurs. In an unchecked context, the operation result is truncated by discarding any high-order bits that don't fit in the destination type. For example, addition wraps from the maximum value to the minimum value. The following example shows the same operation in both a checked and unchecked context:

C#

```
uint a = uint.MaxValue;

unchecked
{
    Console.WriteLine(a + 3);  // output: 2
}

try
{
    checked
    {
        Console.WriteLine(a + 3);
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

> **Note**
>
> The overflow behavior of *user-defined* operators and conversions can differ from the one described in the preceding paragraph. In particular, user-defined checked operators might not throw an exception in a checked context.

For more information, see the Arithmetic overflow and division by zero and User-defined checked operators sections of the Arithmetic operators article.

To specify the overflow-checking context for an expression, you can also use the `checked` and `unchecked` operators, as the following example shows:

C#

```
double a = double.MaxValue;

int b = unchecked((int)a);
Console.WriteLine(b);  // output: -2147483648

try
{
    b = checked((int)a);
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

The `checked` and `unchecked` statements and operators only affect the overflow-checking context for those operations that are *textually* inside the statement block or operator's parentheses, as the following example shows:

C#

```
int Multiply(int a, int b) => a * b;

int factor = 2;

try
{
    checked
    {
        Console.WriteLine(Multiply(factor, int.MaxValue));  // output: -2
    }
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);
}

try
{
    checked
    {
        Console.WriteLine(Multiply(factor, factor * int.MaxValue));
    }
```

```
}
catch (OverflowException e)
{
    Console.WriteLine(e.Message);  // output: Arithmetic operation resulted
in an overflow.
}
```

At the preceding example, the first invocation of the `Multiply` local function shows that the `checked` statement doesn't affect the overflow-checking context within the `Multiply` function as no exception is thrown. At the second invocation of the `Multiply` function, the expression that calculates the second argument of the function is evaluated in a checked context and results in an exception as it's textually inside the block of the `checked` statement.

The behavior of `checked` and `unchecked` depends on the type and the operation. Even for integers, operations like `unchecked(x / 0)` always throw because there's no sensible behavior. Check the behavior for the type and the operation to understand how the `checked` and `unchecked` keywords affect your code.

Numeric types and overflow-checking context

The `checked` and `unchecked` keywords primarily apply to integral types where there's a sensible overflow behavior. The wraparound behavior where `T.MaxValue + 1` becomes `T.MinValue` is sensible in a two's complement value. The represented value isn't *correct* since it can't fit in the storage for the type. Therefore, the bits are representative of the lower n-bits of the full result.

For types like `decimal`, `float`, `double`, and `Half` that represent a more complex value or a one's complement value, wraparound isn't sensible. It can't be used to compute larger or more accurate results, so `unchecked` isn't beneficial.

`float`, `double`, and `Half` have sensible saturating values for `PositiveInfinity` and `NegativeInfinity`, so you can detect overflow in an `unchecked` context. For `decimal`, no such limits exist, and saturating at `MaxValue` can lead to errors or confusion. Operations that use `decimal` throw in both a `checked` and `unchecked` context.

Operations affected by the overflow-checking context

The overflow-checking context affects the following operations:

- The following built-in arithmetic operators: unary `++`, `--`, `-` and binary `+`, `-`, `*`, and `/` operators, when their operands are of an integral type (that is, either integral numeric or char type) or an enum type.
- Explicit numeric conversions between integral types or from float or double to an integral type.

  > **Note**
  >
  > When you convert a `decimal` value to an integral type and the result is outside the range of the destination type, an **OverflowException** is always thrown, regardless of the overflow-checking context.

- Beginning with C# 11, user-defined checked operators and conversions. For more information, see the User-defined checked operators section of the Arithmetic operators article.

Default overflow-checking context

If you don't specify the overflow-checking context, the value of the **CheckForOverflowUnderflow** compiler option defines the default context for nonconstant expressions. By default the value of that option is unset and integral-type arithmetic operations and conversions are executed in an **unchecked** context.

Constant expressions are evaluated by default in a checked context and overflow causes a compile-time error. You can explicitly specify an unchecked context for a constant expression with the `unchecked` statement or operator.

C# language specification

For more information, see the following sections of the C# language specification:

- The checked and unchecked statements
- The checked and unchecked operators
- User-defined checked and unchecked operators - C# 11

See also

- **CheckForOverflowUnderflow** compiler option

fixed statement - pin a variable for pointer operations

Article • 04/10/2023

The `fixed` statement prevents the garbage collector from relocating a moveable variable and declares a pointer to that variable. The address of a fixed, or pinned, variable doesn't change during execution of the statement. You can use the declared pointer only inside the corresponding `fixed` statement. The declared pointer is readonly and can't be modified:

C#

```
unsafe
{
    byte[] bytes = [1, 2, 3];
    fixed (byte* pointerToFirst = bytes)
    {
        Console.WriteLine($"The address of the first array element:
{(long)pointerToFirst:X}.");
        Console.WriteLine($"The value of the first array element:
{*pointerToFirst}.");
    }
}
// Output is similar to:
// The address of the first array element: 2173F80B5C8.
// The value of the first array element: 1.
```

> **Note**
>
> You can use the `fixed` statement only in an unsafe context. The code that contains unsafe blocks must be compiled with the AllowUnsafeBlocks compiler option.

You can initialize the declared pointer as follows:

- With an array, as the example at the beginning of this article shows. The initialized pointer contains the address of the first array element.
- With an address of a variable. Use the address-of & operator, as the following example shows:

  C#

```
unsafe
{
    int[] numbers = [10, 20, 30];
    fixed (int* toFirst = &numbers[0], toLast = &numbers[^1])
    {
        Console.WriteLine(toLast - toFirst);  // output: 2
    }
}
```

Object fields are another example of moveable variables that can be pinned.

When the initialized pointer contains the address of an object field or an array element, the `fixed` statement guarantees that the garbage collector doesn't relocate or dispose of the containing object instance during the execution of the statement body.

- With the instance of the type that implements a method named `GetPinnableReference`. That method must return a `ref` variable of an unmanaged type. The .NET types System.Span<T> and System.ReadOnlySpan<T> make use of this pattern. You can pin span instances, as the following example shows:

C#

```
unsafe
{
    int[] numbers = [10, 20, 30, 40, 50];
    Span<int> interior = numbers.AsSpan()[1..^1];
    fixed (int* p = interior)
    {
        for (int i = 0; i < interior.Length; i++)
        {
            Console.Write(p[i]);
        }
        // output: 203040
    }
}
```

For more information, see the Span<T>.GetPinnableReference() API reference.

- With a string, as the following example shows:

C#

```
unsafe
{
    var message = "Hello!";
    fixed (char* p = message)
    {
```

```
            Console.WriteLine(*p);  // output: H
        }
    }
```

- With a fixed-size buffer.

You can allocate memory on the stack, where it's not subject to garbage collection and therefore doesn't need to be pinned. To do that, use a stackalloc expression.

You can also use the `fixed` keyword to declare a fixed-size buffer.

C# language specification

For more information, see the following sections of the C# language specification:

- The fixed statement
- Fixed and moveable variables

See also

- Unsafe code, pointer types, and function pointers
- Pointer-related operators
- unsafe

The lock statement - ensure exclusive access to a shared resource

Article • 05/07/2024

The `lock` statement acquires the mutual-exclusion lock for a given object, executes a statement block, and then releases the lock. While a lock is held, the thread that holds the lock can again acquire and release the lock. Any other thread is blocked from acquiring the lock and waits until the lock is released. The `lock` statement ensures that at maximum only one thread executes its body at any moment in time.

The `lock` statement takes the following form:

C#

```
lock (x)
{
    // Your code...
}
```

The variable `x` is an expression of System.Threading.Lock type, or a reference type. When `x` is known at compile-time to be of the type System.Threading.Lock, it's precisely equivalent to:

C#

```
using (x.EnterScope())
{
    // Your code...
}
```

The object returned by Lock.EnterScope() is a ref struct that includes a `Dispose()` method. The generated using statement ensures the scope is released even if an exception is thrown with the body of the `lock` statement.

Otherwise, the `lock` statement is precisely equivalent to:

C#

```
object __lockObj = x;
bool __lockWasTaken = false;
try
{
    System.Threading.Monitor.Enter(__lockObj, ref __lockWasTaken);
```

```
    // Your code...
}
finally
{
    if (__lockWasTaken) System.Threading.Monitor.Exit(__lockObj);
}
```

Since the code uses a try-finally statement, the lock is released even if an exception is thrown within the body of a `lock` statement.

You can't use the await expression in the body of a `lock` statement.

Guidelines

Beginning with .NET 9 and C# 13, lock a dedicated object instance of the System.Threading.Lock type for best performance. In addition, the compiler issues a warning if a known `Lock` object is cast to another type and locked. If using an older version of .NET and C#, lock on a dedicated object instance that isn't used for another purpose. Avoid using the same lock object instance for different shared resources, as it might result in deadlock or lock contention. In particular, avoid using the following instances as lock objects:

- `this`, as callers might also lock `this`.
- Type instances, as they might be obtained by the typeof operator or reflection.
- string instances, including string literals, as they might be interned.

Hold a lock for as short time as possible to reduce lock contention.

Example

The following example defines an `Account` class that synchronizes access to its private `balance` field by locking on a dedicated `balanceLock` instance. Using the same instance for locking ensures that two different threads can't update the `balance` field by calling the `Debit` or `Credit` methods simultaneously. The sample uses C# 13 and the new `Lock` object. If you're using an older version of C# or an older .NET library, lock an instance of `object`.

C#

```
using System;
using System.Threading.Tasks;

public class Account
```

```
{
    // Use `object` in versions earlier than C# 13
    private readonly System.Threading.Lock _balanceLock = new();
    private decimal _balance;

    public Account(decimal initialBalance) => _balance = initialBalance;

    public decimal Debit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The debit
amount cannot be negative.");
        }

        decimal appliedAmount = 0;
        lock (_balanceLock)
        {
            if (_balance >= amount)
            {
                _balance -= amount;
                appliedAmount = amount;
            }
        }
        return appliedAmount;
    }

    public void Credit(decimal amount)
    {
        if (amount < 0)
        {
            throw new ArgumentOutOfRangeException(nameof(amount), "The
credit amount cannot be negative.");
        }

        lock (_balanceLock)
        {
            _balance += amount;
        }
    }

    public decimal GetBalance()
    {
        lock (_balanceLock)
        {
            return _balance;
        }
    }
}

class AccountTest
{
    static async Task Main()
    {
        var account = new Account(1000);
```

```
        var tasks = new Task[100];
        for (int i = 0; i < tasks.Length; i++)
        {
            tasks[i] = Task.Run(() => Update(account));
        }
        await Task.WhenAll(tasks);
        Console.WriteLine($"Account's balance is {account.GetBalance()}");
        // Output:
        // Account's balance is 2000
    }

    static void Update(Account account)
    {
        decimal[] amounts = [0, 2, -3, 6, -2, -1, 8, -5, 11, -6];
        foreach (var amount in amounts)
        {
            if (amount >= 0)
            {
                account.Credit(amount);
            }
            else
            {
                account.Debit(Math.Abs(amount));
            }
        }
    }
}
```

C# language specification

For more information, see The lock statement section of the C# language specification.

See also

- System.Threading.Monitor
- System.Threading.SpinLock
- System.Threading.Interlocked
- Overview of synchronization primitives
- Introduction to System.Threading.Channels

using statement - ensure the correct use of disposable objects

10/09/2025

The `using` statement ensures the correct use of an IDisposable instance:

C#

```
var numbers = new List<int>();
using (StreamReader reader = File.OpenText("numbers.txt"))
{
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
}
```

When the control leaves the block of the `using` statement, an acquired IDisposable instance is disposed. In particular, the `using` statement ensures that a disposable instance is disposed even if an exception occurs within the block of the `using` statement. In the preceding example, an opened file is closed after all lines are processed.

Use the `await using` statement to correctly use an IAsyncDisposable instance:

C#

```
await using (var resource = new AsyncDisposableExample())
{
    // Use the resource
}
```

For more information about using of IAsyncDisposable instances, see the Using async disposable section of the Implement a DisposeAsync method article.

You can also use a `using` *declaration* that doesn't require braces:

C#

```
static IEnumerable<int> LoadNumbers(string filePath)
{
    using StreamReader reader = File.OpenText(filePath);
```

```
    var numbers = new List<int>();
    string line;
    while ((line = reader.ReadLine()) is not null)
    {
        if (int.TryParse(line, out int number))
        {
            numbers.Add(number);
        }
    }
    return numbers;
}
```

When declared in a `using` declaration, a local variable is disposed at the end of the scope in which it's declared. In the preceding example, disposal happens at the end of a method.

A variable declared by the `using` statement or declaration is readonly. You cannot reassign it or pass it as a ref or out parameter.

You can declare several instances of the same type in one `using` statement, as the following example shows:

C#

```
using (StreamReader numbersFile = File.OpenText("numbers.txt"), wordsFile =
File.OpenText("words.txt"))
{
    // Process both files
}
```

When you declare several instances in one `using` statement, they are disposed in reverse order of declaration.

You can also use the `using` statement and declaration with an instance of a ref struct that fits the disposable pattern. That is, it has an instance `Dispose` method that's accessible, parameterless, and has a `void` return type.

A `return` inside a `using` block still guarantees disposal, the compiler rewrites it into a `try/finally`, so the resource's `Dispose` is always called before the method actually returns.

The `using` statement can also be of the following form:

C#

```
using (expression)
{
```

```
    // ...
}
```

where `expression` produces a disposable instance. The following example demonstrates that:

C#

```
StreamReader reader = File.OpenText(filePath);

using (reader)
{
    // Process file content
}
```

> ⚠ **Warning**
>
> In the preceding example, after control leaves the `using` statement, a disposable instance remains in scope while it's already disposed. If you use that instance further, you might encounter an exception, for example, ObjectDisposedException. That's why we recommend declaring a disposable variable within the `using` statement or with the `using` declaration.

C# language specification

For more information, see The using statement section of the C# language specification.

See also

- System.IDisposable
- System.IAsyncDisposable
- Using objects that implement IDisposable
- Implement a Dispose method
- Implement a DisposeAsync method
- Use simple 'using' statement (style rule IDE0063)
- using directive

yield statement - provide the next element

08/30/2025

You use the `yield` statement in an iterator to provide the next value or signal the end of an iteration. The `yield` statement has the two following forms:

- `yield return`: to provide the next value in iteration, as the following example shows:

  C#

  ```
  foreach (int i in ProduceEvenNumbers(9))
  {
      Console.Write(i);
      Console.Write(" ");
  }
  // Output: 0 2 4 6 8

  IEnumerable<int> ProduceEvenNumbers(int upto)
  {
      for (int i = 0; i <= upto; i += 2)
      {
          yield return i;
      }
  }
  ```

- `yield break`: to explicitly signal the end of iteration, as the following example shows:

  C#

  ```
  Console.WriteLine(string.Join(" ", TakeWhilePositive(new int[] {2, 3, 4, 5,
  -1, 3, 4})));
  // Output: 2 3 4 5

  Console.WriteLine(string.Join(" ", TakeWhilePositive(new int[] {9, 8, 7})));
  // Output: 9 8 7

  IEnumerable<int> TakeWhilePositive(IEnumerable<int> numbers)
  {
      foreach (int n in numbers)
      {
          if (n > 0)
          {
              yield return n;
          }
          else
          {
              yield break;
          }
  ```

```
    }
}
```

Iteration also finishes when control reaches the end of an iterator.

In the preceding examples, the return type of iterators is IEnumerable<T> (in nongeneric cases, use IEnumerable as the return type of an iterator). You can also use IAsyncEnumerable<T> as the return type of an iterator. That makes an iterator async. Use the await foreach statement to iterate over iterator's result, as the following example shows:

C#

```
await foreach (int n in GenerateNumbersAsync(5))
{
    Console.Write(n);
    Console.Write(" ");
}
// Output: 0 2 4 6 8

async IAsyncEnumerable<int> GenerateNumbersAsync(int count)
{
    for (int i = 0; i < count; i++)
    {
        yield return await ProduceNumberAsync(i);
    }
}

async Task<int> ProduceNumberAsync(int seed)
{
    await Task.Delay(1000);
    return 2 * seed;
}
```

IEnumerator<T> or IEnumerator can also be the return type of an iterator. Use those return types when you implement the `GetEnumerator` method in the following scenarios:

- You design the type that implements IEnumerable<T> or IEnumerable interface.
- You add an instance or extension `GetEnumerator` method to enable iteration over the type's instance with the foreach statement, as the following example shows:

C#

```
public static void Example()
{
    var point = new Point(1, 2, 3);
    foreach (int coordinate in point)
    {
        Console.Write(coordinate);
```

```
            Console.Write(" ");
        }
        // Output: 1 2 3
    }

    public readonly record struct Point(int X, int Y, int Z)
    {
        public IEnumerator<int> GetEnumerator()
        {
            yield return X;
            yield return Y;
            yield return Z;
        }
    }
```

You can't use the `yield` statements in:

- methods with in, ref, or out parameters.
- lambda expressions and anonymous methods.
- unsafe blocks. Before C# 13, `yield` was invalid in any method with an `unsafe` block. Beginning with C# 13, you can use `yield` in methods with `unsafe` blocks, but not in the `unsafe` block.
- `yield return` and `yield break` can not be used in catch and finally blocks, or in try blocks with a corresponding `catch` block. The `yield return` and `yield break` statements can be used in a `try` block with no `catch` blocks, only a `finally` block.

`using` statements in iterators

You can use using statements in iterator methods. Since `using` statements are compiled into `try` blocks with `finally` clauses (and no `catch` blocks), they work correctly with iterators. The disposable resources are properly managed throughout the iterator's execution:

C#

```
Console.WriteLine("=== Using in Iterator Example ===");

// Demonstrate that using statements work correctly in iterators
foreach (string line in ReadLinesFromResource())
{
    Console.WriteLine($"Read: {line}");
    // Simulate processing only first two items
    if (line == "Line 2") break;
}

Console.WriteLine("Iteration stopped early - resource should still be disposed.");

static IEnumerable<string> ReadLinesFromResource()
```

```
{
    Console.WriteLine("Opening resource...");
    using var resource = new StringWriter(); // Use StringWriter as a simple
IDisposable
    resource.WriteLine("Resource initialized");

    // These lines would typically come from the resource (e.g., file, database)
    string[] lines = { "Line 1", "Line 2", "Line 3", "Line 4" };

    foreach (string line in lines)
    {
        Console.WriteLine($"About to yield: {line}");
        yield return line;
        Console.WriteLine($"Resumed after yielding: {line}");
    }

    Console.WriteLine("Iterator completed - using block will dispose resource.");
}
```

As the preceding example shows, the resource acquired in the `using` statement remains available throughout the iterator's execution, even when the iterator suspends and resumes execution at `yield return` statements. The resource is disposed when the iterator completes (either by reaching the end or via `yield break`) or when the iterator itself is disposed (for example, when the caller breaks out of enumeration early).

Execution of an iterator

The call of an iterator doesn't execute it immediately, as the following example shows:

C#

```
var numbers = ProduceEvenNumbers(5);
Console.WriteLine("Caller: about to iterate.");
foreach (int i in numbers)
{
    Console.WriteLine($"Caller: {i}");
}

IEnumerable<int> ProduceEvenNumbers(int upto)
{
    Console.WriteLine("Iterator: start.");
    for (int i = 0; i <= upto; i += 2)
    {
        Console.WriteLine($"Iterator: about to yield {i}");
        yield return i;
        Console.WriteLine($"Iterator: yielded {i}");
    }
    Console.WriteLine("Iterator: end.");
}
// Output:
```

```
// Caller: about to iterate.
// Iterator: start.
// Iterator: about to yield 0
// Caller: 0
// Iterator: yielded 0
// Iterator: about to yield 2
// Caller: 2
// Iterator: yielded 2
// Iterator: about to yield 4
// Caller: 4
// Iterator: yielded 4
// Iterator: end.
```

As the preceding example shows, when you start to iterate over an iterator's result, an iterator is executed until the first `yield return` statement is reached. Then, the execution of an iterator is suspended and the caller gets the first iteration value and processes it. On each subsequent iteration, the execution of an iterator resumes after the `yield return` statement that caused the previous suspension and continues until the next `yield return` statement is reached. The iteration completes when control reaches the end of an iterator or a `yield break` statement.

C# language specification

For more information, see The yield statement section of the C# language specification.

See also

- Iterators
- Iterate through collections in C#
- foreach
- await foreach

C# Special Characters

10/07/2025

Special characters are predefined, contextual characters that modify the program element (a literal string, an identifier, or an attribute name) to which they're prepended. C# supports the following special characters:

- @, the verbatim identifier character.
- $, the interpolated string character.
- """, A sequence of three or more `"` characters provides the delimiters for a raw string literal.
- _, a `_` character represents a *discard*, which is a placeholder for an unused variable.

This section only includes those tokens that aren't operators. See the operators section for all operators.

Code comments - `//` and `/*`. `*/`

Article • 12/13/2022

C# supports two different forms of comments. Single line comments start with `//` and end at the end of that line of code. Multiline comments start with `/*` and end with `*/`. The following code shows an example of each:

C#

```
// This is a single line comment.

/* This could be a summary of all the
   code that's in this class.
   You might add multiple paragraphs, or links to pages
   like https://learn.microsoft.com/dotnet/csharp.

   You could even include emojis. This example is 🔥
   Then, when you're done, close with
   */
```

The multi-line comment can also be used to insert text in a line of code. Because these comments have an explicit closing character, you can include more executable code after the comment:

C#

```
public static int Add(int left, int right)
{
    return left /* first operand */ + right /* second operand */;
}
```

The single line comment can appear after executable code on the same line. The comment ends at the end of the text line:

C#

```
return source++; // increment the source.
```

Some comments start with three slashes: `///`. *Triple-slash comments* are *XML documentation comments*. The compiler reads these to produce human documentation. You can read more about XML doc comments in the section on triple-slash comments.

Discard - A `_` acts as a placeholder for a variable

Article • 12/21/2024

The `_` character serves as a *discard*, which is a placeholder for an unused variable.

There are two uses for the *discard* token:

1. To declare an unused variable. A discard can't be read or accessed.
    - Unused `out` arguments: `var r = M(out int _, out var _, out _);`
    - Unused lambda expression parameters: `Action<int> _ => WriteMessage();`
    - Unused deconstruction arguments: `(int _, var answer) = M();`
2. To match any expression in a discard pattern. You can add a `_` pattern to satisfy exhaustiveness requirements.

The `_` token is a valid identifier in C#. The `_` token is interpreted as a discard only when no valid identifier named `_` is found in scope.

A discard can't be read as a variable. The compiler reports an error if your code reads a discard. The compiler can avoid allocating the storage for a discard in some situations where that is safe.

See also

- Tuples
- Deconstruction
- Discard pattern

String interpolation using `$`

10/08/2025

The `$` character identifies a string literal as an *interpolated string*. An interpolated string is a string literal that might contain *interpolation expressions*. When an interpolated string is resolved to a result string, the compiler replaces items with interpolation expressions by the string representations of the expression results.

String interpolation provides a more readable, convenient syntax to format strings. It's easier to read than string composite formatting. The following example uses both features to produce the same output:

C#

```
var name = "Mark";
var date = DateTime.Now;

// Composite formatting:
Console.WriteLine("Hello, {0}! Today is {1}, it's {2:HH:mm} now.", name,
date.DayOfWeek, date);
// String interpolation:
Console.WriteLine($"Hello, {name}! Today is {date.DayOfWeek}, it's {date:HH:mm}
now.");
// Both calls produce the same output that is similar to:
// Hello, Mark! Today is Wednesday, it's 19:40 now.
```

You can use an interpolated string to initialize a constant string. You can do that only if all interpolation expressions within the interpolated string are constant strings as well.

Structure of an interpolated string

To identify a string literal as an interpolated string, prepend it with the `$` symbol. You can't have any white space between the `$` and the `"` that starts a string literal.

The structure of an item with an interpolation expression is as follows:

C#

```
{<interpolationExpression>[,<width>][:<formatString>]}
```

Elements in square brackets are optional. The following table describes each element:

Expand table

Element	Description
`interpolationExpression`	The expression that produces a result to be formatted. When the expression is `null`, the output is the empty string (String.Empty).
`width`	The constant expression whose value defines the minimum number of characters in the string representation of the expression result. If positive, the string representation is right-aligned; if negative, left-aligned. For more information, see the Width component section of the Composite formatting article.
`formatString`	A format string supported by the type of the expression result. For more information, see the Format string component section of the Composite formatting article.

The following example uses optional formatting components described in the preceding table:

C#

```
Console.WriteLine($"|{"Left",-7}|{"Right",7}|");

const int FieldWidthRightAligned = 20;
Console.WriteLine($"{Math.PI,FieldWidthRightAligned} - default formatting of the
pi number");
Console.WriteLine($"{Math.PI,FieldWidthRightAligned:F3} - display only three
decimal digits of the pi number");
// Output is:
// |Left    |  Right|
//     3.14159265358979 - default formatting of the pi number
//                3.142 - display only three decimal digits of the pi number
```

Beginning with C# 11, you can use new-lines within an interpolation expression to make the expression's code more readable. The following example shows how new-lines can improve the readability of an expression involving pattern matching:

C#

```
string message = $"The usage policy for {safetyScore} is {
    safetyScore switch
    {
        > 90 => "Unlimited usage",
        > 80 => "General usage, with daily safety check",
        > 70 => "Issues must be addressed within 1 week",
        > 50 => "Issues must be addressed within 1 day",
        _ => "Issues must be addressed before continued use",
    }
    }";
```

Interpolated raw string literals

Beginning with C# 11, you can use an interpolated raw string literal, as the following example shows:

C#

```
int X = 2;
int Y = 3;

var pointMessage = $"""The point "{X}, {Y}" is {Math.Sqrt(X * X + Y * Y):F3} from
the origin""";

Console.WriteLine(pointMessage);
// Output is:
// The point "2, 3" is 3.606 from the origin
```

To embed `{` and `}` characters in the result string, start an interpolated raw string literal with multiple `$` characters. When you do that, any sequence of `{` or `}` characters shorter than the number of `$` characters is embedded in the result string. To enclose any interpolation expression within that string, you need to use the same number of braces as the number of `$` characters, as the following example shows:

C#

```
int X = 2;
int Y = 3;

var pointMessage = $$"""{The point {{{X}}, {{Y}}} is {{Math.Sqrt(X * X + Y *
Y):F3}} from the origin}""";
Console.WriteLine(pointMessage);
// Output is:
// {The point {2, 3} is 3.606 from the origin}
```

In the preceding example, an interpolated raw string literal starts with two `$` characters. You need to put every interpolation expression between double braces (`{{` and `}}`). A single brace is embedded into a result string. If you need to embed repeated `{` or `}` characters into a result string, use an appropriately greater number of `$` characters to designate an interpolated raw string literal. If the string literal has more repeated braces than the number of `$` characters, the `{` and `}` characters are grouped from inside to outside. In the preceding example, the literal `The point {{{X}}, {{Y}}}` interprets `{{X}}` and `{{Y}}` as interpolated expressions. The outer `{` and `}` are included verbatim in the output string.

> **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.

Special characters

To include a brace, "{" or "}", in the text produced by an interpolated string, use two braces, "{{" or "}}". For more information, see the Escaping braces section of the Composite formatting article.

As the colon (":") has special meaning in an interpolation expression item, to use a conditional operator in an interpolation expression, enclose that expression in parentheses.

The following example shows how to include a brace in a result string. It also shows how to use a conditional operator:

C#

```
string name = "Horace";
int age = 34;
Console.WriteLine($"He asked, \"Is your name {name}?\", but didn't wait for a
reply :-{{");
Console.WriteLine($"{name} is {age} year{(age == 1 ? "" : "s")} old.");
// Output is:
// He asked, "Is your name Horace?", but didn't wait for a reply :-{
// Horace is 34 years old.
```

An interpolated verbatim string starts with both the `$` and `@` characters. You can use `$` and `@` in any order: both `$@"..."` and `@$"..."` are valid interpolated verbatim strings. For more information about verbatim strings, see the string and verbatim identifier articles.

Culture-specific formatting

By default, an interpolated string uses the current culture defined by the CultureInfo.CurrentCulture property for all formatting operations.

To resolve an interpolated string to a culture-specific result string, use the String.Create(IFormatProvider, DefaultInterpolatedStringHandler) method, which is available beginning with .NET 6. The following example shows how to do that:

C#

```
double speedOfLight = 299792.458;

System.Globalization.CultureInfo.CurrentCulture =
System.Globalization.CultureInfo.GetCultureInfo("nl-NL");
string messageInCurrentCulture = $"The speed of light is {speedOfLight:N3} km/s.";

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = string.Create(
    specificCulture, $"The speed of light is {speedOfLight:N3} km/s.");

string messageInInvariantCulture = string.Create(
    System.Globalization.CultureInfo.InvariantCulture, $"The speed of light is
{speedOfLight:N3} km/s.");

Console.WriteLine($"{System.Globalization.CultureInfo.CurrentCulture,-10}
{messageInCurrentCulture}");
Console.WriteLine($"{specificCulture,-10} {messageInSpecificCulture}");
Console.WriteLine($"{"Invariant",-10} {messageInInvariantCulture}");
// Output is:
// nl-NL      The speed of light is 299.792,458 km/s.
// en-IN      The speed of light is 2,99,792.458 km/s.
// Invariant  The speed of light is 299,792.458 km/s.
```

In .NET 5 and earlier versions of .NET, use implicit conversion of an interpolated string to a FormattableString instance. Then, you can use an instance FormattableString.ToString(IFormatProvider) method or a static FormattableString.Invariant method to produce a culture-specific result string. The following example shows how to do that:

C#

```
double speedOfLight = 299792.458;
FormattableString message = $"The speed of light is {speedOfLight:N3} km/s.";

var specificCulture = System.Globalization.CultureInfo.GetCultureInfo("en-IN");
string messageInSpecificCulture = message.ToString(specificCulture);
Console.WriteLine(messageInSpecificCulture);
// Output:
// The speed of light is 2,99,792.458 km/s.

string messageInInvariantCulture = FormattableString.Invariant(message);
Console.WriteLine(messageInInvariantCulture);
// Output is:
// The speed of light is 299,792.458 km/s.
```

For more information about custom formatting, see the Custom formatting with ICustomFormatter section of the Formatting types in .NET article.

Other resources

If you're new to string interpolation, see the String interpolation in C# interactive tutorial. That tutorial demonstrates how to use interpolated strings to produce formatted strings.

Compilation of interpolated strings

The compiler checks if an interpolated string is assigned to a type that satisfies the *interpolated string handler pattern*. An *interpolated string handler* is a type that converts the interpolated string into a result string. When an interpolated string has the type `string`, it's processed by the System.Runtime.CompilerServices.DefaultInterpolatedStringHandler. For the example of a custom interpolated string handler, see the Write a custom string interpolation handler tutorial. Use of an interpolated string handler is an advanced scenario, typically required for performance reasons.

> **Note**
>
> One side effect of interpolated string handlers is that a custom handler, including System.Runtime.CompilerServices.DefaultInterpolatedStringHandler, might not evaluate all the interpolation expressions within the interpolated string under all conditions. That means side effects of those expressions might not occur.

If an interpolated string has the type `string`, it's typically transformed into a String.Format method call. The compiler can replace String.Format with String.Concat if the analyzed behavior would be equivalent to concatenation.

If an interpolated string has the type IFormattable or FormattableString, the compiler generates a call to the FormattableStringFactory.Create method.

C# language specification

For more information, see the Interpolated string expressions section of the C# language specification and the following new feature specifications:

- Improved interpolated strings
- C# 11 - Raw string literals
- C# 11 - New-lines in string interpolations

See also

- C# special characters
- Strings
- Standard numeric format strings
- Composite formatting
- String.Format
- Simplify interpolation (style rule IDE0071)
- String interpolation in C# 10 and .NET 6 (.NET blog)

Verbatim text - `@` in variables, attributes, and string literals

Article • 03/22/2023

The `@` special character serves as a verbatim identifier. You use it in the following ways:

1. To indicate that a string literal is to be interpreted verbatim. The `@` character in this instance defines a *verbatim string literal*. Simple escape sequences (such as `"\\"` for a backslash), hexadecimal escape sequences (such as `"\x0041"` for an uppercase A), and Unicode escape sequences (such as `"\u0041"` for an uppercase A) are interpreted literally. Only a quote escape sequence (`""`) isn't interpreted literally; it produces one double quotation mark. Additionally, in case of a verbatim interpolated string brace escape sequences (`{{` and `}}`) aren't interpreted literally; they produce single brace characters. The following example defines two identical file paths, one by using a regular string literal and the other by using a verbatim string literal. This is one of the more common uses of verbatim string literals.

   C#

   ```
   string filename1 = @"c:\documents\files\u0066.txt";
   string filename2 = "c:\\documents\\files\\u0066.txt";

   Console.WriteLine(filename1);
   Console.WriteLine(filename2);
   // The example displays the following output:
   //     c:\documents\files\u0066.txt
   //     c:\documents\files\u0066.txt
   ```

   The following example illustrates the effect of defining a regular string literal and a verbatim string literal that contain identical character sequences.

   C#

   ```
   string s1 = "He said, \"This is the last \u0063hance\x0021\"";
   string s2 = @"He said, ""This is the last \u0063hance\x0021""";

   Console.WriteLine(s1);
   Console.WriteLine(s2);
   // The example displays the following output:
   //     He said, "This is the last chance!"
   //     He said, "This is the last \u0063hance\x0021"
   ```

2. To use C# keywords as identifiers. The `@` character prefixes a code element that the compiler is to interpret as an identifier rather than a C# keyword. The following example uses the `@` character to define an identifier named `for` that it uses in a `for` loop.

```
C#
```

```
string[] @for = { "John", "James", "Joan", "Jamie" };
for (int ctr = 0; ctr < @for.Length; ctr++)
{
   Console.WriteLine($"Here is your gift, {@for[ctr]}!");
}
// The example displays the following output:
//     Here is your gift, John!
//     Here is your gift, James!
//     Here is your gift, Joan!
//     Here is your gift, Jamie!
```

3. To enable the compiler to distinguish between attributes in cases of a naming conflict. An attribute is a class that derives from Attribute. Its type name typically includes the suffix **Attribute**, although the compiler doesn't enforce this convention. The attribute can then be referenced in code either by its full type name (for example, `[InfoAttribute]` or its shortened name (for example, `[Info]`). However, a naming conflict occurs if two shortened attribute type names are identical, and one type name includes the **Attribute** suffix but the other doesn't. For example, the following code fails to compile because the compiler can't determine whether the `Info` or `InfoAttribute` attribute is applied to the `Example` class. For more information, see CS1614.

```
C#
```

```
using System;

[AttributeUsage(AttributeTargets.Class)]
public class Info : Attribute
{
   private string information;

   public Info(string info)
   {
      information = info;
   }
}

[AttributeUsage(AttributeTargets.Method)]
public class InfoAttribute : Attribute
{
   private string information;
```

```
   public InfoAttribute(string info)
   {
      information = info;
   }
}

[Info("A simple executable.")] // Generates compiler error CS1614.
Ambiguous Info and InfoAttribute.
// Prepend '@' to select 'Info' ([@Info("A simple executable.")]).
Specify the full name 'InfoAttribute' to select it.
public class Example
{
   [InfoAttribute("The entry point.")]
   public static void Main()
   {
   }
}
```

See also

- C# Special Characters

Raw string literal text - `"""` in string literals

10/08/2025

A raw string literal starts and ends with a minimum of three double quote (`"`) characters:

```C#
var singleLine = """This is a "raw string literal". It can contain characters like \, ' and ".""";
```

Raw string literals can span multiple lines:

```C#
var xml = """
        <element attr="content">
            <body>
            </body>
        </element>
        """;
```

The following rules govern the interpretation of a multi-line raw string literal:

- The opening quotes must be the last non-whitespace characters on their line, and the closing quotes must be the first non-whitespace characters on their line.
- Any whitespace to the left of the closing quotes is removed from all lines of the raw string literal.
- Any whitespace following the opening quotes on the same line is ignored.
- Whitespace only lines following the opening quote are included in the string literal.
- If a whitespace precedes the end delimiter on the same line, the exact number and kind of whitespace characters (for example, spaces vs. tabs) must exist at the beginning of each content line. Specifically, a space doesn't match a horizontal tab, and vice versa.
- The newline before the closing quotes isn't included in the literal string.

> **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.

You might need to create a raw string literal that has three or more consecutive double-quote characters. Raw string literals can start and end with a sequence of at least three double-quote characters. When your string literal contains three consecutive double-quotes, you start and end the raw string literal with four double quote characters:

C#

```
var moreQuotes = """ As you can see,"""Raw string literals""" can start and end
with more than three double-quotes when needed.""";
```

If you need to start or end a raw string literal with quote characters, use the multi-line format:

C#

```
var MultiLineQuotes = """
                """Raw string literals""" can start and end with more than three
double-quotes when needed.
                """;
```

Raw string literals can also be combined with interpolated strings to embed the `{` and `}` characters in the output string. You use multiple `$` characters in an interpolated raw string literal to embed `{` and `}` characters in the output string without escaping them.

The raw string literal's content must not contain a set of contiguous `"` characters whose length is equal to or greater than the raw string literal delimiter length. For example, the strings `""" """ """` and `""""""" """""" """"" """" """ """""""` are well-formed. However, the strings `""" """ """` and `""" """" """` are ill-formed

Raw string literals were introduced in C# 11.

See also

- C# special characters
- C# string interpolation
- Raw string literals feature specification

Assembly level attributes interpreted by the C# compiler

Article • 07/30/2024

Most attributes are applied to specific language elements such as classes or methods; however, some attributes are global—they apply to an entire assembly or module. For example, the AssemblyVersionAttribute attribute can be used to embed version information into an assembly, like this:

C#

```
[assembly: AssemblyVersion("1.0.0.0")]
```

Global attributes appear in the source code after any top level `using` directives and before any type, module, or namespace declarations. Global attributes can appear in multiple source files, but the files must be compiled in a single compilation pass. Visual Studio adds global attributes to the AssemblyInfo.cs file in .NET Framework projects. These attributes aren't added to .NET Core projects.

Assembly attributes are values that provide information about an assembly. They fall into the following categories:

- Assembly identity attributes
- Informational attributes
- Assembly manifest attributes

Assembly identity attributes

Three attributes (with a strong name, if applicable) determine the identity of an assembly: name, version, and culture. These attributes form the full name of the assembly and are required when you reference it in code. You can set an assembly's version and culture using attributes. However, the name value is set by the compiler, the Visual Studio IDE in the Assembly Information Dialog Box, or the Assembly Linker (Al.exe) when the assembly is created. The assembly name is based on the assembly manifest. The AssemblyFlagsAttribute attribute specifies whether multiple copies of the assembly can coexist.

The following table shows the identity attributes.

Expand table

Attribute	Purpose
AssemblyVersionAttribute	Specifies the version of an assembly.
AssemblyCultureAttribute	Specifies which culture the assembly supports.
AssemblyFlagsAttribute	Specifies a bitwise combination of flags for an assembly, describing just-in-time (JIT) compiler options, whether the assembly is retargetable, and whether it has a full or tokenized public key.

Informational attributes

You use informational attributes to provide more company or product information for an assembly. The following table shows the informational attributes defined in the System.Reflection namespace.

Expand table

Attribute	Purpose
AssemblyProductAttribute	Specifies a product name for an assembly manifest.
AssemblyTrademarkAttribute	Specifies a trademark for an assembly manifest.
AssemblyInformationalVersionAttribute	Specifies an informational version for an assembly manifest.
AssemblyCompanyAttribute	Specifies a company name for an assembly manifest.
AssemblyCopyrightAttribute	Defines a custom attribute that specifies a copyright for an assembly manifest.
AssemblyFileVersionAttribute	Sets a specific version number for the Win32 file version resource.
CLSCompliantAttribute	Indicates whether the assembly is compliant with the Common Language Specification (CLS).

Assembly manifest attributes

You can use assembly manifest attributes to provide information in the assembly manifest. The attributes include title, description, default alias, and configuration. The following table shows the assembly manifest attributes defined in the System.Reflection namespace.

Expand table

Attribute	Purpose
AssemblyTitleAttribute	Specifies an assembly title for an assembly manifest.
AssemblyDescriptionAttribute	Specifies an assembly description for an assembly manifest.
AssemblyConfigurationAttribute	Specifies an assembly configuration (such as retail or debug) for an assembly manifest.
AssemblyDefaultAliasAttribute	Defines a friendly default alias for an assembly manifest

Determine caller information using attributes interpreted by the C# compiler

Article • 06/15/2022

Using info attributes, you obtain information about the caller to a method. You obtain the file path of the source code, the line number in the source code, and the member name of the caller. To obtain member caller information, you use attributes that are applied to optional parameters. Each optional parameter specifies a default value. The following table lists the Caller Info attributes that are defined in the System.Runtime.CompilerServices namespace:

Expand table

Attribute	Description	Type
CallerFilePathAttribute	Full path of the source file that contains the caller. The full path is the path at compile time.	`String`
CallerLineNumberAttribute	Line number in the source file from which the method is called.	`Integer`
CallerMemberNameAttribute	Method name or property name of the caller.	`String`
CallerArgumentExpressionAttribute	String representation of the argument expression.	`String`

This information helps you with tracing and debugging, and helps you to create diagnostic tools. The following example shows how to use caller info attributes. On each call to the `TraceMessage` method, the caller information is inserted for the arguments to the optional parameters.

C#

```
public void DoProcessing()
{
    TraceMessage("Something happened.");
}

public void TraceMessage(string message,
        [CallerMemberName] string memberName = "",
        [CallerFilePath] string sourceFilePath = "",
        [CallerLineNumber] int sourceLineNumber = 0)
{
```

```
    Trace.WriteLine("message: " + message);
    Trace.WriteLine("member name: " + memberName);
    Trace.WriteLine("source file path: " + sourceFilePath);
    Trace.WriteLine("source line number: " + sourceLineNumber);
}

// Sample Output:
//  message: Something happened.
//  member name: DoProcessing
//  source file path: c:\Visual Studio
Projects\CallerInfoCS\CallerInfoCS\Form1.cs
//  source line number: 31
```

You specify an explicit default value for each optional parameter. You can't apply caller info attributes to parameters that aren't specified as optional. The caller info attributes don't make a parameter optional. Instead, they affect the default value that's passed in when the argument is omitted. Caller info values are emitted as literals into the Intermediate Language (IL) at compile time. Unlike the results of the StackTrace property for exceptions, the results aren't affected by obfuscation. You can explicitly supply the optional arguments to control the caller information or to hide caller information.

Member names

You can use the `CallerMemberName` attribute to avoid specifying the member name as a `String` argument to the called method. By using this technique, you avoid the problem that **Rename Refactoring** doesn't change the `String` values. This benefit is especially useful for the following tasks:

- Using tracing and diagnostic routines.
- Implementing the INotifyPropertyChanged interface when binding data. This interface allows the property of an object to notify a bound control that the property has changed. The control can display the updated information. Without the `CallerMemberName` attribute, you must specify the property name as a literal.

The following chart shows the member names that are returned when you use the `CallerMemberName` attribute.

Expand table

Calls occur within	Member name result
Method, property, or event	The name of the method, property, or event from which the call originated.
Constructor	The string ".ctor"

Calls occur within	Member name result
Static constructor	The string ".cctor"
Finalizer	The string "Finalize"
User-defined operators or conversions	The generated name for the member, for example, "op_Addition".
Attribute constructor	The name of the method or property to which the attribute is applied. If the attribute is any element within a member (such as a parameter, a return value, or a generic type parameter), this result is the name of the member that's associated with that element.
No containing member (for example, assembly-level or attributes that are applied to types)	The default value of the optional parameter.

Argument expressions

You use the System.Runtime.CompilerServices.CallerArgumentExpressionAttribute when you want the expression passed as an argument. Diagnostic libraries may want to provide more details about the *expressions* passed to arguments. By providing the expression that triggered the diagnostic, in addition to the parameter name, developers have more details about the condition that triggered the diagnostic. That extra information makes it easier to fix.

The following example shows how you can provide detailed information about the argument when it's invalid:

C#

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

You would invoke it as shown in the following example:

C#

```
public void Operation(Action func)
{
    Utilities.ValidateArgument(nameof(func), func is not null);
    func();
}
```

The expression used for `condition` is injected by the compiler into the `message` argument. When a developer calls `Operation` with a `null` argument, the following message is stored in the `ArgumentException`:

```
text

Argument failed validation: <func is not null>
```

This attribute enables you to write diagnostic utilities that provide more details. Developers can more quickly understand what changes are needed. You can also use the CallerArgumentExpressionAttribute to determine what expression was used as the receiver for extension methods. The following method samples a sequence at regular intervals. If the sequence has fewer elements than the frequency, it reports an error:

```
C#

public static IEnumerable<T> Sample<T>(this IEnumerable<T> sequence, int
frequency,
    [CallerArgumentExpression(nameof(sequence))] string? message = null)
{
    if (sequence.Count() < frequency)
        throw new ArgumentException($"Expression doesn't have enough
elements: {message}", nameof(sequence));
    int i = 0;
    foreach (T item in sequence)
    {
        if (i++ % frequency == 0)
            yield return item;
    }
}
```

The previous example uses the nameof operator for the parameter `sequence`. That feature is available in C# 11. Before C# 11, you'll need to type the name of the parameter as a string. You could call this method as follows:

```
C#

sample = Enumerable.Range(0, 10).Sample(100);
```

The preceding example would throw an ArgumentException whose message is the following text:

```text
Expression doesn't have enough elements: Enumerable.Range(0, 10) (Parameter
'sequence')
```

See also

- Named and Optional Arguments
- System.Reflection
- Attribute
- Attributes

Attributes for null-state static analysis interpreted by the C# compiler

Article • 05/05/2023

In a nullable enabled context, the compiler performs static analysis of code to determine the *null-state* of all reference type variables:

- *not-null*: Static analysis determines that a variable has a non-null value.
- *maybe-null*: Static analysis can't determine that a variable is assigned a non-null value.

These states enable the compiler to provide warnings when you may dereference a null value, throwing a System.NullReferenceException. These attributes provide the compiler with semantic information about the *null-state* of arguments, return values, and object members based on the state of arguments and return values. The compiler provides more accurate warnings when your APIs have been properly annotated with this semantic information.

This article provides a brief description of each of the nullable reference type attributes and how to use them.

Let's start with an example. Imagine your library has the following API to retrieve a resource string. This method was originally compiled in a *nullable oblivious* context:

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

The preceding example follows the familiar `Try*` pattern in .NET. There are two reference parameters for this API: the `key` and the `message`. This API has the following rules relating to the *null-state* of these parameters:

- Callers shouldn't pass `null` as the argument for `key`.
- Callers can pass a variable whose value is `null` as the argument for `message`.

- If the `TryGetMessage` method returns `true`, the value of `message` isn't null. If the return value is `false`, the value of `message` is null.

The rule for `key` can be expressed succinctly: `key` should be a non-nullable reference type. The `message` parameter is more complex. It allows a variable that is `null` as the argument, but guarantees, on success, that the `out` argument isn't `null`. For these scenarios, you need a richer vocabulary to describe the expectations. The `NotNullWhen` attribute, described below describes the *null-state* for the argument used for the `message` parameter.

> **Note**
>
> Adding these attributes gives the compiler more information about the rules for your API. When calling code is compiled in a nullable enabled context, the compiler will warn callers when they violate those rules. These attributes don't enable more checks on your implementation.

Expand table

Attribute	Category	Meaning
AllowNull	Precondition	A non-nullable parameter, field, or property may be null.
DisallowNull	Precondition	A nullable parameter, field, or property should never be null.
MaybeNull	Postcondition	A non-nullable parameter, field, property, or return value may be null.
NotNull	Postcondition	A nullable parameter, field, property, or return value will never be null.
MaybeNullWhen	Conditional postcondition	A non-nullable argument may be null when the method returns the specified `bool` value.
NotNullWhen	Conditional postcondition	A nullable argument won't be null when the method returns the specified `bool` value.
NotNullIfNotNull	Conditional postcondition	A return value, property, or argument isn't null if the argument for the specified parameter isn't null.
MemberNotNull	Method and property helper methods	The listed member won't be null when the method returns.

Attribute	Category	Meaning
MemberNotNullWhen	Method and property helper methods	The listed member won't be null when the method returns the specified `bool` value.
DoesNotReturn	Unreachable code	A method or property never returns. In other words, it always throws an exception.
DoesNotReturnIf	Unreachable code	This method or property never returns if the associated `bool` parameter has the specified value.

The preceding descriptions are a quick reference to what each attribute does. The following sections describe the behavior and meaning of these attributes more thoroughly.

Preconditions: `AllowNull` and `DisallowNull`

Consider a read/write property that never returns `null` because it has a reasonable default value. Callers pass `null` to the set accessor when setting it to that default value. For example, consider a messaging system that asks for a screen name in a chat room. If none is provided, the system generates a random name:

C#

```
public string ScreenName
{
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName;
```

When you compile the preceding code in a nullable oblivious context, everything is fine. Once you enable nullable reference types, the `ScreenName` property becomes a non-nullable reference. That's correct for the `get` accessor: it never returns `null`. Callers don't need to check the returned property for `null`. But now setting the property to `null` generates a warning. To support this type of code, you add the System.Diagnostics.CodeAnalysis.AllowNullAttribute attribute to the property, as shown in the following code:

C#

```
[AllowNull]
public string ScreenName
{
```

```
    get => _screenName;
    set => _screenName = value ?? GenerateRandomScreenName();
}
private string _screenName = GenerateRandomScreenName();
```

You may need to add a `using` directive for System.Diagnostics.CodeAnalysis to use this and other attributes discussed in this article. The attribute is applied to the property, not the `set` accessor. The `AllowNull` attribute specifies *pre-conditions*, and only applies to arguments. The `get` accessor has a return value, but no parameters. Therefore, the `AllowNull` attribute only applies to the `set` accessor.

The preceding example demonstrates what to look for when adding the `AllowNull` attribute on an argument:

1. The general contract for that variable is that it shouldn't be `null`, so you want a non-nullable reference type.
2. There are scenarios for a caller to pass `null` as the argument, though they aren't the most common usage.

Most often you'll need this attribute for properties, or `in`, `out`, and `ref` arguments. The `AllowNull` attribute is the best choice when a variable is typically non-null, but you need to allow `null` as a precondition.

Contrast that with scenarios for using `DisallowNull`: You use this attribute to specify that an argument of a nullable reference type shouldn't be `null`. Consider a property where `null` is the default value, but clients can only set it to a non-null value. Consider the following code:

C#

```
public string ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new
ArgumentNullException(nameof(value), "Cannot set to null");
}
string _comment;
```

The preceding code is the best way to express your design that the `ReviewComment` could be `null`, but can't be set to `null`. Once this code is nullable aware, you can express this concept more clearly to callers using the System.Diagnostics.CodeAnalysis.DisallowNullAttribute:

C#

```
[DisallowNull]
public string? ReviewComment
{
    get => _comment;
    set => _comment = value ?? throw new
ArgumentNullException(nameof(value), "Cannot set to null");
}
string? _comment;
```

In a nullable context, the `ReviewComment` `get` accessor could return the default value of `null`. The compiler warns that it must be checked before access. Furthermore, it warns callers that, even though it could be `null`, callers shouldn't explicitly set it to `null`. The `DisallowNull` attribute also specifies a *pre-condition*, it doesn't affect the `get` accessor. You use the `DisallowNull` attribute when you observe these characteristics about:

1. The variable could be `null` in core scenarios, often when first instantiated.
2. The variable shouldn't be explicitly set to `null`.

These situations are common in code that was originally *null oblivious*. It may be that object properties are set in two distinct initialization operations. It may be that some properties are set only after some asynchronous work has completed.

The `AllowNull` and `DisallowNull` attributes enable you to specify that preconditions on variables may not match the nullable annotations on those variables. These provide more detail about the characteristics of your API. This additional information helps callers use your API correctly. Remember you specify preconditions using the following attributes:

- AllowNull: A non-nullable argument may be null.
- DisallowNull: A nullable argument should never be null.

Postconditions: `MaybeNull` and `NotNull`

Suppose you have a method with the following signature:

C#

```
public Customer FindCustomer(string lastName, string firstName)
```

You've likely written a method like this to return `null` when the name sought wasn't found. The `null` clearly indicates that the record wasn't found. In this example, you'd

likely change the return type from `Customer` to `Customer?`. Declaring the return value as a nullable reference type specifies the intent of this API clearly:

```
C#

public Customer? FindCustomer(string lastName, string firstName)
```

For reasons covered under Generics nullability that technique may not produce the static analysis that matches your API. You may have a generic method that follows a similar pattern:

```
C#

public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

The method returns `null` when the sought item isn't found. You can clarify that the method returns `null` when an item isn't found by adding the `MaybeNull` annotation to the method return:

```
C#

[return: MaybeNull]
public T Find<T>(IEnumerable<T> sequence, Func<T, bool> predicate)
```

The preceding code informs callers that the return value *may* actually be null. It also informs the compiler that the method may return a `null` expression even though the type is non-nullable. When you have a generic method that returns an instance of its type parameter, `T`, you can express that it never returns `null` by using the `NotNull` attribute.

You can also specify that a return value or an argument isn't null even though the type is a nullable reference type. The following method is a helper method that throws if its first argument is `null`:

```
C#

public static void ThrowWhenNull(object value, string valueExpression = "")
{
    if (value is null) throw new ArgumentNullException(nameof(value),
valueExpression);
}
```

You could call this routine as follows:

C#

```
public static void LogMessage(string? message)
{
    ThrowWhenNull(message, $"{nameof(message)} must not be null");

    Console.WriteLine(message.Length);
}
```

After enabling null reference types, you want to ensure that the preceding code compiles without warnings. When the method returns, the `value` parameter is guaranteed to be not null. However, it's acceptable to call `ThrowWhenNull` with a null reference. You can make `value` a nullable reference type, and add the `NotNull` post-condition to the parameter declaration:

C#

```
public static void ThrowWhenNull([NotNull] object? value, string
valueExpression = "")
{
    _ = value ?? throw new ArgumentNullException(nameof(value),
valueExpression);
    // other logic elided
```

The preceding code expresses the existing contract clearly: Callers can pass a variable with the `null` value, but the argument is guaranteed to never be null if the method returns without throwing an exception.

You specify unconditional postconditions using the following attributes:

- MaybeNull: A non-nullable return value may be null.
- NotNull: A nullable return value will never be null.

Conditional post-conditions: `NotNullWhen`, `MaybeNullWhen`, and `NotNullIfNotNull`

You're likely familiar with the `string` method String.IsNullOrEmpty(String). This method returns `true` when the argument is null or an empty string. It's a form of null-check: Callers don't need to null-check the argument if the method returns `false`. To make a method like this nullable aware, you'd set the argument to a nullable reference type, and add the `NotNullWhen` attribute:

C#

```
bool IsNullOrEmpty([NotNullWhen(false)] string? value)
```

That informs the compiler that any code where the return value is `false` doesn't need null checks. The addition of the attribute informs the compiler's static analysis that `IsNullOrEmpty` performs the necessary null check: when it returns `false`, the argument isn't `null`.

C#

```
string? userInput = GetUserInput();
if (!string.IsNullOrEmpty(userInput))
{
    int messageLength = userInput.Length; // no null check needed.
}
// null check needed on userInput here.
```

The String.IsNullOrEmpty(String) method will be annotated as shown above for .NET Core 3.0. You may have similar methods in your codebase that check the state of objects for null values. The compiler won't recognize custom null check methods, and you'll need to add the annotations yourself. When you add the attribute, the compiler's static analysis knows when the tested variable has been null checked.

Another use for these attributes is the `Try*` pattern. The postconditions for `ref` and `out` arguments are communicated through the return value. Consider this method shown earlier (in a nullable disabled context):

C#

```
bool TryGetMessage(string key, out string message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message != null;
}
```

The preceding method follows a typical .NET idiom: the return value indicates if `message` was set to the found value or, if no message is found, to the default value. If the method returns `true`, the value of `message` isn't null; otherwise, the method sets `message` to null.

In a nullable enabled context, you can communicate that idiom using the `NotNullWhen` attribute. When you annotate parameters for nullable reference types, make `message` a `string?` and add an attribute:

C#

```
bool TryGetMessage(string key, [NotNullWhen(true)] out string? message)
{
    if (_messageMap.ContainsKey(key))
        message = _messageMap[key];
    else
        message = null;
    return message is not null;
}
```

In the preceding example, the value of `message` is known to be not null when `TryGetMessage` returns `true`. You should annotate similar methods in your codebase in the same way: the arguments could equal `null`, and are known to be not null when the method returns `true`.

There's one final attribute you may also need. Sometimes the null state of a return value depends on the null state of one or more arguments. These methods will return a non-null value whenever certain arguments aren't `null`. To correctly annotate these methods, you use the `NotNullIfNotNull` attribute. Consider the following method:

C#

```
string GetTopLevelDomainFromFullUrl(string url)
```

If the `url` argument isn't null, the output isn't `null`. Once nullable references are enabled, you need to add more annotations if your API may accept a null argument. You could annotate the return type as shown in the following code:

C#

```
string? GetTopLevelDomainFromFullUrl(string? url)
```

That also works, but will often force callers to implement extra `null` checks. The contract is that the return value would be `null` only when the argument `url` is `null`. To express that contract, you would annotate this method as shown in the following code:

C#

```
[return: NotNullIfNotNull(nameof(url))]
string? GetTopLevelDomainFromFullUrl(string? url)
```

The previous example uses the nameof operator for the parameter `url`. That feature is available in C# 11. Before C# 11, you'll need to type the name of the parameter as a string. The return value and the argument have both been annotated with the `?` indicating that either could be `null`. The attribute further clarifies that the return value won't be null when the `url` argument isn't `null`.

You specify conditional postconditions using these attributes:

- MaybeNullWhen: A non-nullable argument may be null when the method returns the specified `bool` value.
- NotNullWhen: A nullable argument won't be null when the method returns the specified `bool` value.
- NotNullIfNotNull: A return value isn't null if the argument for the specified parameter isn't null.

Helper methods: `MemberNotNull` and `MemberNotNullWhen`

These attributes specify your intent when you've refactored common code from constructors into helper methods. The C# compiler analyzes constructors and field initializers to make sure that all non-nullable reference fields have been initialized before each constructor returns. However, the C# compiler doesn't track field assignments through all helper methods. The compiler issues warning `CS8618` when fields aren't initialized directly in the constructor, but rather in a helper method. You add the MemberNotNullAttribute to a method declaration and specify the fields that are initialized to a non-null value in the method. For example, consider the following example:

C#

```
public class Container
{
    private string _uniqueIdentifier; // must be initialized.
    private string? _optionalMessage;

    public Container()
    {
        Helper();
    }

    public Container(string message)
    {
        Helper();
        _optionalMessage = message;
```

```
    }

    [MemberNotNull(nameof(_uniqueIdentifier))]
    private void Helper()
    {
        _uniqueIdentifier = DateTime.Now.Ticks.ToString();
    }
}
```

You can specify multiple field names as arguments to the `MemberNotNull` attribute constructor.

The MemberNotNullWhenAttribute has a `bool` argument. You use `MemberNotNullWhen` in situations where your helper method returns a `bool` indicating whether your helper method initialized fields.

Stop nullable analysis when called method throws

Some methods, typically exception helpers or other utility methods, always exit by throwing an exception. Or, a helper may throw an exception based on the value of a Boolean argument.

In the first case, you can add the DoesNotReturnAttribute attribute to the method declaration. The compiler's *null-state* analysis doesn't check any code in a method that follows a call to a method annotated with `DoesNotReturn`. Consider this method:

C#

```
[DoesNotReturn]
private void FailFast()
{
    throw new InvalidOperationException();
}

public void SetState(object containedField)
{
    if (containedField is null)
    {
        FailFast();
    }

    // containedField can't be null:
    _field = containedField;
}
```

The compiler doesn't issue any warnings after the call to `FailFast`.

In the second case, you add the System.Diagnostics.CodeAnalysis.DoesNotReturnIfAttribute attribute to a Boolean parameter of the method. You can modify the previous example as follows:

C#

```
private void FailFastIf([DoesNotReturnIf(true)] bool isNull)
{
    if (isNull)
    {
        throw new InvalidOperationException();
    }
}

public void SetFieldState(object? containedField)
{
    FailFastIf(containedField == null);
    // No warning: containedField can't be null here:
    _field = containedField;
}
```

When the value of the argument matches the value of the `DoesNotReturnIf` constructor, the compiler doesn't perform any *null-state* analysis after that method.

Summary

Adding nullable reference types provides an initial vocabulary to describe your APIs expectations for variables that could be `null`. The attributes provide a richer vocabulary to describe the null state of variables as preconditions and postconditions. These attributes more clearly describe your expectations and provide a better experience for the developers using your APIs.

As you update libraries for a nullable context, add these attributes to guide users of your APIs to the correct usage. These attributes help you fully describe the null-state of arguments and return values.

- AllowNull: A non-nullable field, parameter, or property may be null.
- DisallowNull: A nullable field, parameter, or property should never be null.
- MaybeNull: A non-nullable field, parameter, property, or return value may be null.
- NotNull: A nullable field, parameter, property, or return value will never be null.
- MaybeNullWhen: A non-nullable argument may be null when the method returns the specified `bool` value.

- NotNullWhen: A nullable argument won't be null when the method returns the specified `bool` value.
- NotNullIfNotNull: A parameter, property, or return value isn't null if the argument for the specified parameter isn't null.
- DoesNotReturn: A method or property never returns. In other words, it always throws an exception.
- DoesNotReturnIf: This method or property never returns if the associated `bool` parameter has the specified value.

Miscellaneous attributes interpreted by the C# compiler

Article • 01/29/2025

There are several attributes that can be applied to elements in your code that add semantic meaning to those elements:

- Conditional: Make execution of a method dependent on a preprocessor identifier.
- Obsolete: Mark a type or member for (potential) future removal.
- Deprecated: (Windows Foundation) Mark a type or member for (potential) future removal.
- Experimental: Mark a type or member as experimental.
- SetsRequiredMembers: Indicate that a constructor sets all required properties.
- AttributeUsage: Declare the language elements where an attribute can be applied.
- AsyncMethodBuilder: Declare an async method builder type.
- InterpolatedStringHandler: Define an interpolated string builder for a known scenario.
- ModuleInitializer: Declare a method that initializes a module.
- SkipLocalsInit: Elide the code that initializes local variable storage to 0.
- UnscopedRef: Declare that a `ref` variable normally interpreted as `scoped` should be treated as unscoped.
- OverloadResolutionPriority: Add a tiebreaker attribute to influence overload resolution for possibly ambiguous overloads.
- EnumeratorCancellation: Specify the parameter for a cancellation token in an async enumerator.
- CollectionBuilder: Specify the builder method for a collection type when a collection expression is converted to that collection type.
- InlineArray: Specify that a `struct` type is an *inline array*.
- IUnknownConstant: Specifies how a missing argument should be supplied for a default parameter.
- IDispatchConstant: Specifies how a missing argument should be supplied for a default parameter.

The compiler uses those semantic meanings to alter its output and report possible mistakes by developers using your code.

`Conditional` attribute

The `Conditional` attribute makes the execution of a method dependent on a preprocessing identifier. The `Conditional` attribute is an alias for ConditionalAttribute, and can be applied to a method or an attribute class.

In the following example, `Conditional` is applied to a method to enable or disable the display of program-specific diagnostic information:

C#

```
#define TRACE_ON
using System.Diagnostics;

namespace AttributeExamples;

public class Trace
{
    [Conditional("TRACE_ON")]
    public static void Msg(string msg)
    {
        Console.WriteLine(msg);
    }
}

public class TraceExample
{
    public static void Main()
    {
        Trace.Msg("Now in Main...");
        Console.WriteLine("Done.");
    }
}
```

If the `TRACE_ON` identifier isn't defined, the trace output isn't displayed. Explore for yourself in the interactive window.

The `Conditional` attribute is often used with the `DEBUG` identifier to enable trace and logging features for debug builds but not in release builds, as shown in the following example:

C#

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

When a method marked conditional is called, the presence or absence of the specified preprocessing symbol determines whether the compiler includes or omits calls to the

method. If the symbol is defined, the call is included; otherwise, the call is omitted. A conditional method must be a method in a class or struct declaration and must have a `void` return type. Using `Conditional` is cleaner, more elegant, and less error-prone than enclosing methods inside `#if…#endif` blocks.

If a method has multiple `Conditional` attributes, compiler includes calls to the method if one or more conditional symbols are defined (the symbols are logically linked together by using the OR operator). In the following example, the presence of either `A` or `B` results in a method call:

C#

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Using `Conditional` with attribute classes

The `Conditional` attribute can also be applied to an attribute class definition. In the following example, the custom attribute `Documentation` adds information to the metadata if `DEBUG` is defined.

C#

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

`Obsolete` and `Deprecated` attribute

The `Obsolete` attribute marks a code element as no longer recommended for use. Use of an entity marked obsolete generates a warning or an error. The `Obsolete` attribute is a single-use attribute and can be applied to any entity that allows attributes. `Obsolete` is an alias for ObsoleteAttribute.

In the following example, the `Obsolete` attribute is applied to class `A` and to method `B.OldMethod`. Because the second argument of the attribute constructor applied to `B.OldMethod` is set to `true`, this method causes a compiler error, whereas using class `A` produces a warning. Calling `B.NewMethod`, however, produces no warning or error. For example, when you use it with the previous definitions, the following code generates two warnings and one error:

C#

```
namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }

        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}
```

The string provided as the first argument to the attribute constructor is displayed as part of the warning or error. Two warnings for class `A` are generated: one for the declaration of the class reference, and one for the class constructor. The `Obsolete` attribute can be used without arguments, but including an explanation what to use instead is recommended. You can use constant string interpolation and the `nameof` operator to ensure the names match:

```
C#
```

```
public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}
```

The Windows Foundation Metadata libraries use the Windows.Foundation.Metadata.DeprecatedAttribute instead of the `ObsoleteAttribute`.

`Experimental` attributes

Beginning in C# 12, types, methods, and 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 declared in an assembly or module marked with the `Experimental` attribute are experimental. The compiler issues a warning if you access any of them. You can disable these warnings to pilot an experimental feature.

> ⚠ **Warning**
>
> Experimental features are subject to changes. The APIs can change, or they can be removed in future updates. Including experimental features is a way for library authors to get feedback on ideas and concepts for future development. Use extreme caution when using any feature marked as experimental. You can learn more about how APIs are marked as experimental in our article on **preview APIs**.

You can read more details about the `Experimental` attribute in the feature specification.

The Windows Foundation Metadata libraries use the Windows.Foundation.Metadata.ExperimentalAttribute, which predates C# 12.

`SetsRequiredMembers` attribute

The `SetsRequiredMembers` attribute informs the compiler that a constructor sets all `required` members in that class or struct. The compiler assumes any constructor with the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute attribute initializes all `required` members. Any code that invokes such a constructor doesn't need object initializers to set required members. Adding the `SetsRequiredMembers` attribute is primarily useful for positional records and primary constructors.

`AttributeUsage` attribute

The `AttributeUsage` attribute determines how a custom attribute class can be used. AttributeUsageAttribute is an attribute you apply to custom attribute definitions. The `AttributeUsage` attribute enables you to control:

- Which program elements the attribute can be applied to. Unless you restrict its usage, an attribute can be applied to any of the following program elements:
  - Assembly
  - Module
  - Field
  - Event
  - Method
  - Parameter
  - Property
  - Return
  - Type
- Whether an attribute can be applied to a single program element multiple times.
- Whether derived classes inherit attributes.

The default settings look like the following example when applied explicitly:

C#

```
[AttributeUsage(AttributeTargets.All,
                  AllowMultiple = false,
                  Inherited = true)]
class NewAttribute : Attribute { }
```

In this example, the `NewAttribute` class can be applied to any supported program element. But it can be applied only once to each entity. Derived classes inherit the attribute applied to a base class.

The AllowMultiple and Inherited arguments are optional, so the following code has the same effect:

C#

```
[AttributeUsage(AttributeTargets.All)]
class NewAttribute : Attribute { }
```

The first AttributeUsageAttribute argument must be one or more elements of the AttributeTargets enumeration. Multiple target types can be linked together with the OR operator, like the following example shows:

C#

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Attributes can be applied to either the property or the backing field for an automatically implemented property. The attribute applies to the property, unless you specify the `field` specifier on the attribute. Both are shown in the following example:

C#

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; } = string.Empty;

    // Attribute attached to backing field:
    [field: NewPropertyOrField]
    public string Description { get; set; } = string.Empty;
}
```

If the AllowMultiple argument is `true`, then the resulting attribute can be applied more than once to a single entity, as shown in the following example:

C#

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }
```

```
[MultiUse, MultiUse]
class Class2 { }
```

In this case, `MultiUseAttribute` can be applied repeatedly because `AllowMultiple` is set to `true`. Both formats shown for applying multiple attributes are valid.

If Inherited is `false`, then derived classes don't inherit the attribute from an attributed base class. For example:

C#

```
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

In this case, `NonInheritedAttribute` isn't applied to `DClass` via inheritance.

You can also use these keywords to specify where an attribute should be applied. For example, you can use the `field:` specifier to add an attribute to the backing field of an automatically implemented property. Or you can use the `field:`, `property:` or `param:` specifier to apply an attribute to any of the elements generated from a positional record. For an example, see Positional syntax for property definition.

`AsyncMethodBuilder` attribute

You add the System.Runtime.CompilerServices.AsyncMethodBuilderAttribute attribute to a type that can be an async return type. The attribute specifies the type that builds the async method implementation when the specified type is returned from an async method. The `AsyncMethodBuilder` attribute can be applied to a type that:

- Has an accessible `GetAwaiter` method.
- The object returned by the `GetAwaiter` method implements the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface.

The constructor to the `AsyncMethodBuilder` attribute specifies the type of the associated builder. The builder must implement the following accessible members:

- A static `Create()` method that returns the type of the builder.
- A readable `Task` property that returns the async return type.

- A `void SetException(Exception)` method that sets the exception when a task faults.
- A `void SetResult()` or `void SetResult(T result)` method that marks the task as completed and optionally sets the task's result
- A `Start` method with the following API signature:

```
C#

void Start<TStateMachine>(ref TStateMachine stateMachine)
          where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

- An `AwaitOnCompleted` method with the following signature:

```
C#

public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref TAwaiter
awaiter, ref TStateMachine stateMachine)
    where TAwaiter : System.Runtime.CompilerServices.INotifyCompletion
    where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

- An `AwaitUnsafeOnCompleted` method with the following signature:

```
C#

      public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>(ref
TAwaiter awaiter, ref TStateMachine stateMachine)
          where TAwaiter :
System.Runtime.CompilerServices.ICriticalNotifyCompletion
          where TStateMachine :
System.Runtime.CompilerServices.IAsyncStateMachine
```

You can learn about async method builders by reading about the following builders supplied by .NET:

- System.Runtime.CompilerServices.AsyncTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>

The `AsyncMethodBuilder` attribute can be applied to an async method to override the builder for that type.

`InterpolatedStringHandler` and `InterpolatedStringHandlerArguments` attributes

You use these attributes to specify that a type is an *interpolated string handler*. The .NET 6 library already includes System.Runtime.CompilerServices.DefaultInterpolatedStringHandler for scenarios where you use an interpolated string as the argument for a `string` parameter. You might have other instances where you want to control how interpolated strings are processed. You apply the System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute to the type that implements your handler. You apply the System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute to parameters of that type's constructor.

You can learn more about building an interpolated string handler in the feature specification for interpolated string improvements.

`ModuleInitializer` attribute

The `ModuleInitializer` attribute marks a method that the runtime calls when the assembly loads. `ModuleInitializer` is an alias for ModuleInitializerAttribute.

The `ModuleInitializer` attribute can only be applied to a method that:

- Is static.
- Is parameterless.
- Returns `void`.
- Is accessible from the containing module, that is, `internal` or `public`.
- Isn't a generic method.
- Isn't contained in a generic class.
- Isn't a local function.

The `ModuleInitializer` attribute can be applied to multiple methods. In that case, the order in which the runtime calls them is deterministic but not specified.

The following example illustrates use of multiple module initializer methods. The `Init1` and `Init2` methods run before `Main`, and each adds a string to the `Text` property. So when `Main` runs, the `Text` property already has strings from both initializer methods.

C#

```
using System;
```

```
internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

C#

```
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

Source code generators sometimes need to generate initialization code. Module initializers provide a standard place for that code. In most other cases, you should write a static constructor instead of a module initializer.

`SkipLocalsInit` attribute

The `SkipLocalsInit` attribute prevents the compiler from setting the `.locals init` flag when emitting to metadata. The `SkipLocalsInit` attribute is a single-use attribute and can be applied to a method, a property, a class, a struct, an interface, or a module, but not to an assembly. `SkipLocalsInit` is an alias for SkipLocalsInitAttribute.

The `.locals init` flag causes the CLR to initialize all of the local variables declared in a method to their default values. Since the compiler also makes sure that you never use a variable before assigning some value to it, `.locals init` is typically not necessary. However, the extra zero-initialization might have measurable performance impact in some scenarios, such as when you use stackalloc to allocate an array on the stack. In

those cases, you can add the `SkipLocalsInit` attribute. If applied to a method directly, the attribute affects that method and all its nested functions, including lambdas and local functions. If applied to a type or module, it affects all methods nested inside. This attribute doesn't affect abstract methods, but it does affect code generated for the implementation.

This attribute requires the AllowUnsafeBlocks compiler option. This requirement signals that in some cases code could view unassigned memory (for example, reading from uninitialized stack-allocated memory).

The following example illustrates the effect of `SkipLocalsInit` attribute on a method that uses `stackalloc`. The method displays whatever was in memory when the array of integers was allocated.

C#

```
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
//-1271631451
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

To try this code yourself, set the `AllowUnsafeBlocks` compiler option in your *.csproj* file:

XML

```
<PropertyGroup>
  ...
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

`UnscopedRef` attribute

The `UnscopedRef` attribute marks a variable declaration as unscoped, meaning the reference is allowed to escape.

You add this attribute where the compiler treats a `ref` as implicitly `scoped`:

- The `this` parameter for `struct` instance methods.
- `ref` parameters that refer to `ref struct` types.
- `out` parameters.

Applying the System.Diagnostics.CodeAnalysis.UnscopedRefAttribute marks the element as unscoped.

`OverloadResolutionPriority` attribute

The OverloadResolutionPriorityAttribute enables library authors to prefer one overload over another when two overloads can be ambiguous. Its primary use case is for library authors to write better performing overloads while still supporting existing code without breaks.

For example, you might add a new overload that uses ReadOnlySpan<T> to reduce memory allocations:

C#

```
[OverloadResolutionPriority(1)]
public void M(params ReadOnlySpan<int> s) => Console.WriteLine("Span");
// Default overload resolution priority of 0
public void M(params int[] a) => Console.WriteLine("Array");
```

Overload resolution considers the two methods equally good for some argument types. For an argument of `int[]`, it prefers the first overload. To get the compiler to prefer the `ReadOnlySpan` version, you can increase the priority of that overload. The following example shows the effect of adding the attribute:

C#

```
var d = new OverloadExample();
int[] arr = [1, 2, 3];
d.M(1, 2, 3, 4); // Prints "Span"
d.M(arr); // Prints "Span" when PriorityAttribute is applied
```

```
d.M([1, 2, 3, 4]); // Prints "Span"
d.M(1, 2, 3, 4); // Prints "Span"
```

All overloads with a lower priority than the highest overload priority are removed from the set of applicable methods. Methods without this attribute have the overload priority set to the default of zero. Library authors should use this attribute as a last resort when adding a new and better method overload. Library authors should have a deep understanding of how Overload resolution impacts choosing the better method. Otherwise, unexpected errors can result.

EnumeratorCancellation attribute

The System.Runtime.CompilerServices.EnumeratorCancellationAttribute attribute specifies which parameter should receive the cancellation token from the System.Collections.Generic.IAsyncEnumerable<T>.GetAsyncEnumerator(CancellationToken) API. It's part of the infrastructure for the async streams feature.

CollectionBuilder attribute

The System.Runtime.CompilerServices.CollectionBuilderAttribute attribute specifies a method that builds an instance of a collection type from a collection expression. You use this attribute to specify a method that builds the collection. The compiler generates code to call that method when a collection expression is converted to that type.

InlineArray attribute

The System.Runtime.CompilerServices.InlineArrayAttribute attribute marks a type as an inline array. You can learn more about this feature in the article on `structs`, in the section on inline arrays.

IUnknownConstant and IDispatchConstant attributes

The System.Runtime.CompilerServices.IUnknownConstantAttribute and System.Runtime.CompilerServices.IDispatchConstantAttribute attributes can be added to default parameters to indicate that a missing argument should be supplied as `new UnknownWrapper(null)` or `new DispatchWrapper(null)`.

See also

- Attribute
- System.Reflection
- Attributes
- Reflection

Custom attributes that generate flags or options in the Intermediate Language (IL) output

Article • 01/31/2025

You add these attributes to your code for the compiler to emit a specified Intermediate Language (IL) modifier. These attributes instruct the compiler to include the corresponding IL modifier in the output.

Expand table

Attribute	Modifier	Comments
System.Runtime.InteropServices.ComImportAttribute	`import`	
System.Runtime.InteropServices.DllImportAttribute	`pinvokeimpl`	You can add options listed in the constructor.
System.Runtime.InteropServices.FieldOffsetAttribute	`.field`	This sets the field offset for memory layout.
MarshalAsAttribute	`marshal`	You can set options listed in the constructor.
System.Runtime.CompilerServices.MethodImplAttribute	`flag`	Constructor arguments specify specific named flags such as `aggressiveinlining` or `forwardref`. These flags also specify the `native`, `managed`, or `optil` modifiers for the System.Runtime.CompilerServices.MethodCodeType field.
System.NonSerializedAttribute	`notserialized`	
System.Runtime.InteropServices.OptionalAttribute	`[opt]`	
System.Runtime.InteropServices.PreserveSigAttribute	`preservesig`	
System.SerializableAttribute	`serializable`	
System.Runtime.InteropServices.StructLayoutAttribute	`auto`, `sequential`, or `explicit`	Layout options can be set using the parameters.
System.Runtime.CompilerServices.IndexerNameAttribute		You add this attribute to an indexer to set a different method name. By default, indexers are compiled to a property named `Item`. You can specify a different name using this attribute.

Some of these custom attributes are applied using other C# syntax rather than adding the attribute to your source code.

Expand table

Attribute	Comments
System.Runtime.InteropServices.DefaultParameterValueAttribute	Specifies the default value for the parameter. Use the default parameter syntax.

Attribute	Comments
System.Runtime.InteropServices.InAttribute	Specifies the IL `[in]` modifier. Use the in or ref readonly modifiers.
System.Runtime.InteropServices.OutAttribute	Specifies the IL `[out]` modifier. Use the out modifier.
System.Runtime.CompilerServices.SpecialNameAttribute	Specifies the IL `specialname` modifier. The compiler automatically adds this modifier for methods that require it.
System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute	This attribute is required for the `delegate*` feature. The compiler adds it to any delegate* that requires its use. However, you must add this attribute to any method declaration when that method is assigned to a function pointer.

The following attributes are generally disallowed in C# source. They're listed here to aid library authors who use reflection, and to ensure you don't create custom attributes with the same fully qualified name.

Expand table

Attribute	Comments
System.Runtime.CompilerServices.CompilerFeatureRequiredAttribute	Prevents downlevel compilers from using metadata it can't safely understand.
System.Runtime.CompilerServices.DecimalConstantAttribute	Encodes `const decimal` fields. The runtime doesn't support `decimal` values as constant values.
System.Reflection.DefaultMemberAttribute	Encodes indexers with System.Runtime.CompilerServices.IndexerNameAttribute. This attribute notes the default indexer when its name is different than `Item`. This attribute is allowed in source.
System.Runtime.CompilerServices.DynamicAttribute	Encodes whether a type in a signature is `dynamic` (versus `object`).
System.Runtime.CompilerServices.ExtensionAttribute	This attribute notes extension methods. The compiler also places this attribute on the containing classes.
System.Runtime.CompilerServices.FixedBufferAttribute	This attribute specifies `fixed` struct fields.
System.Runtime.CompilerServices.IsByRefLikeAttribute	This attribute specifies a `ref` struct.
System.Runtime.CompilerServices.IsReadOnlyAttribute	This attribute indicates that a parameter has the `in` modifier. It distinguishes `in` parameters from `readonly ref` or `[In] ref`.
System.Runtime.CompilerServices.RequiresLocationAttribute	This attribute indicates that a parameter has the `readonly ref` modifier. It distinguishes `readonly ref` from `in` or `[In] ref`.
System.Runtime.CompilerServices.IsUnmanagedAttribute	This attribute specifies the `unmanaged` constraint on a type parameter.
System.Runtime.CompilerServices.NullableAttribute, System.Runtime.CompilerServices.NullableContextAttribute, System.Runtime.CompilerServices.NullablePublicOnlyAttribute	These attributes encode nullable annotations in your source code.

Attribute	Comments
System.ParamArrayAttribute	This attribute encodes the `params` modifier on array parameters.
System.Runtime.CompilerServices.ParamCollectionAttribute	This attribute encodes the `params` modifier on non-array parameters.
System.Runtime.CompilerServices.RefSafetyRulesAttribute	This attribute specifies the C# version that is required in order to understand ref safety annotations in the assembly. Ref safety rules evolve as C# gets new features.
System.Runtime.CompilerServices.RequiredMemberAttribute	This attribute indicates that the `required` modifier was placed on a member declaration. It's the encoding of the required members language feature.
System.Runtime.CompilerServices.TupleElementNamesAttribute	This attribute encodes tuple element names used in signatures.

In addition, the compiler can generate a declaration for other attributes used internally. The compiler generates these attributes in the System.Runtime.CompilerServices namespace for its own use. Some aren't in the .NET Runtime libraries. Instead, the compiler synthesizes a definition for an `internal` type declaration in any assembly where the attribute is needed.

Unsafe code, pointer types, and function pointers

Article • 02/06/2025

Most of the C# code you write is "verifiably safe code." *Verifiably safe code* means .NET tools can verify that the code is safe. In general, safe code doesn't directly access memory using pointers. It also doesn't allocate raw memory. It creates managed objects instead.

C# supports an unsafe context, in which you can write *unverifiable* code. In an `unsafe` context, code can use pointers, allocate and free blocks of memory, and call methods using function pointers. Unsafe code in C# isn't necessarily dangerous; it's just code whose safety can't be verified.

Unsafe code has the following properties:

- Methods, types, and code blocks can be defined as unsafe.
- In some cases, unsafe code can increase an application's performance by removing array bounds checks.
- Unsafe code is required when you call native functions that require pointers.
- Using unsafe code introduces security and stability risks.
- The code that contains unsafe blocks must be compiled with the **AllowUnsafeBlocks** compiler option.

Pointer types

In an unsafe context, a type can be a pointer type, in addition to a value type, or a reference type. A pointer type declaration takes one of the following forms:

C#

```
type* identifier;
void* identifier; //allowed but not recommended
```

The type specified before the `*` in a pointer type is called the **referent type**.

Pointer types don't inherit from object and no conversions exist between pointer types and `object`. Also, boxing and unboxing don't support pointers. However, you can convert between different pointer types and between pointer types and integral types.

When you declare multiple pointers in the same declaration, you write the asterisk (`*`) together with the underlying type only. It isn't used as a prefix to each pointer name. For example:

```
C#
```

```
int* p1, p2, p3;   // Ok
int *p1, *p2, *p3;   // Invalid in C#
```

The garbage collector doesn't keep track of whether an object is being pointed to by any pointer types. If the referrant is an object in the managed heap (including local variables captured by lambda expressions or anonymous delegates), the object must be pinned for as long as the pointer is used.

The value of the pointer variable of type `MyType*` is the address of a variable of type `MyType`. The following are examples of pointer type declarations:

- `int* p`: `p` is a pointer to an integer.
- `int** p`: `p` is a pointer to a pointer to an integer.
- `int*[] p`: `p` is a single-dimensional array of pointers to integers.
- `char* p`: `p` is a pointer to a char.
- `void* p`: `p` is a pointer to an unknown type.

The pointer indirection operator `*` can be used to access the contents at the location pointed to by the pointer variable. For example, consider the following declaration:

```
C#
```

```
int* myVariable;
```

The expression `*myVariable` denotes the `int` variable found at the address contained in `myVariable`.

There are several examples of pointers in the articles on the fixed statement. The following example uses the `unsafe` keyword and the `fixed` statement, and shows how to increment an interior pointer. You can paste this code into the Main function of a console application to run it. These examples must be compiled with the **AllowUnsafeBlocks** compiler option set.

```
C#
```

```
// Normal pointer to an object.
int[] a = [10, 20, 30, 40, 50];
```

```
// Must be in unsafe code to use interior pointers.
unsafe
{
    // Must pin object on heap so that it doesn't move while using interior
pointers.
    fixed (int* p = &a[0])
    {
        // p is pinned as well as object, so create another pointer to show
incrementing it.
        int* p2 = p;
        Console.WriteLine(*p2);
        // Incrementing p2 bumps the pointer by four bytes due to its type
...
        p2 += 1;
        Console.WriteLine(*p2);
        p2 += 1;
        Console.WriteLine(*p2);
        Console.WriteLine("--------");
        Console.WriteLine(*p);
        // Dereferencing p and incrementing changes the value of a[0] ...
        *p += 1;
        Console.WriteLine(*p);
        *p += 1;
        Console.WriteLine(*p);
    }
}

Console.WriteLine("--------");
Console.WriteLine(a[0]);

/*
Output:
10
20
30
--------
10
11
12
--------
12
*/
```

You can't apply the indirection operator to a pointer of type `void*`. However, you can use a cast to convert a void pointer to any other pointer type, and vice versa.

A pointer can be `null`. Applying the indirection operator to a null pointer causes an implementation-defined behavior.

Passing pointers between methods can cause undefined behavior. Consider a method that returns a pointer to a local variable through an `in`, `out`, or `ref` parameter or as the

function result. If the pointer was set in a fixed block, the variable to which it points might no longer be fixed.

The following table lists the operators and statements that can operate on pointers in an unsafe context:

Expand table

Operator/Statement	Use
`*`	Performs pointer indirection.
`->`	Accesses a member of a struct through a pointer.
`[]`	Indexes a pointer.
`&`	Obtains the address of a variable.
`++` and `--`	Increments and decrements pointers.
`+` and `-`	Performs pointer arithmetic.
`==`, `!=`, `<`, `>`, `<=`, and `>=`	Compares pointers.
stackalloc	Allocates memory on the stack.
fixed statement	Temporarily fixes a variable so that its address can be found.

For more information about pointer-related operators, see Pointer-related operators.

Any pointer type can be implicitly converted to a `void*` type. Any pointer type can be assigned the value `null`. Any pointer type can be explicitly converted to any other pointer type using a cast expression. You can also convert any integral type to a pointer type, or any pointer type to an integral type. These conversions require an explicit cast.

The following example converts an `int*` to a `byte*`. Notice that the pointer points to the lowest addressed byte of the variable. When you successively increment the result, up to the size of `int` (4 bytes), you can display the remaining bytes of the variable.

C#

```
int number = 1024;

unsafe
{
    // Convert to byte:
    byte* p = (byte*)&number;

    System.Console.Write("The 4 bytes of the integer:");
```

```
    // Display the 4 bytes of the int variable:
    for (int i = 0 ; i < sizeof(int) ; ++i)
    {
        System.Console.Write(" {0:X2}", *p);
        // Increment the pointer:
        p++;
    }
    System.Console.WriteLine();
    System.Console.WriteLine($"The value of the integer: {number}");

    /* Output:
        The 4 bytes of the integer: 00 04 00 00
        The value of the integer: 1024
    */
}
```

Fixed-size buffers

You can use the `fixed` keyword to create a buffer with a fixed-size array in a data structure. Fixed-size buffers are useful when you write methods that interoperate with data sources from other languages or platforms. The fixed-size buffer can take any attributes or modifiers that are allowed for regular struct members. The only restriction is that the array type must be `bool`, `byte`, `char`, `short`, `int`, `long`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double`.

C#

```
private fixed char name[30];
```

In safe code, a C# struct that contains an array doesn't contain the array elements. The struct contains a reference to the elements instead. You can embed an array of fixed size in a struct when it's used in an unsafe code block.

The size of the following `struct` doesn't depend on the number of elements in the array, since `pathName` is a reference:

C#

```
public struct PathArray
{
    public char[] pathName;
    private int reserved;
}
```

A struct can contain an embedded array in unsafe code. In the following example, the `fixedBuffer` array has a fixed size. You use a fixed statement to get a pointer to the first element. You access the elements of the array through this pointer. The `fixed` statement pins the `fixedBuffer` instance field to a specific location in memory.

C#

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}

internal unsafe class Example
{
    public Buffer buffer = default;
}

private static void AccessEmbeddedArray()
{
    var example = new Example();

    unsafe
    {
        // Pin the buffer to a fixed location in memory.
        fixed (char* charPtr = example.buffer.fixedBuffer)
        {
            *charPtr = 'A';
        }
        // Access safely through the index:
        char c = example.buffer.fixedBuffer[0];
        Console.WriteLine(c);

        // Modify through the index:
        example.buffer.fixedBuffer[0] = 'B';
        Console.WriteLine(example.buffer.fixedBuffer[0]);
    }
}
```

The size of the 128 element `char` array is 256 bytes. Fixed-size char buffers always take 2 bytes per character, regardless of the encoding. This array size is the same even when char buffers are marshaled to API methods or structs with `CharSet = CharSet.Auto` or `CharSet = CharSet.Ansi`. For more information, see CharSet.

The preceding example demonstrates accessing `fixed` fields without pinning. Another common fixed-size array is the bool array. The elements in a `bool` array are always 1 byte in size. `bool` arrays aren't appropriate for creating bit arrays or buffers.

Fixed-size buffers are compiled with the System.Runtime.CompilerServices.UnsafeValueTypeAttribute, which instructs the common language runtime (CLR) that a type contains an unmanaged array that can potentially overflow. Memory allocated using stackalloc also automatically enables buffer overrun detection features in the CLR. The previous example shows how a fixed-size buffer could exist in an `unsafe struct`.

C#

```
internal unsafe struct Buffer
{
    public fixed char fixedBuffer[128];
}
```

The compiler-generated C# for `Buffer` is attributed as follows:

C#

```
internal struct Buffer
{
    [StructLayout(LayoutKind.Sequential, Size = 256)]
    [CompilerGenerated]
    [UnsafeValueType]
    public struct <fixedBuffer>e__FixedBuffer
    {
        public char FixedElementField;
    }

    [FixedBuffer(typeof(char), 128)]
    public <fixedBuffer>e__FixedBuffer fixedBuffer;
}
```

Fixed-size buffers differ from regular arrays in the following ways:

- May only be used in an `unsafe` context.
- May only be instance fields of structs.
- They're always vectors, or one-dimensional arrays.
- The declaration should include the length, such as `fixed char id[8]`. You can't use `fixed char id[]`.

How to use pointers to copy an array of bytes

The following example uses pointers to copy bytes from one array to another.

This example uses the unsafe keyword, which enables you to use pointers in the `Copy` method. The fixed statement is used to declare pointers to the source and destination arrays. The `fixed` statement *pins* the location of the source and destination arrays in memory so that garbage collection doesn't move the arrays. The memory blocks for the arrays are unpinned when the `fixed` block is completed. Because the `Copy` method in this example uses the `unsafe` keyword, it must be compiled with the **AllowUnsafeBlocks** compiler option.

This example accesses the elements of both arrays using indices rather than a second unmanaged pointer. The declaration of the `pSource` and `pTarget` pointers pins the arrays.

C#

```
static unsafe void Copy(byte[] source, int sourceOffset, byte[] target,
    int targetOffset, int count)
{
    // If either array is not instantiated, you cannot complete the copy.
    if ((source == null) || (target == null))
    {
        throw new System.ArgumentException("source or target is null");
    }

    // If either offset, or the number of bytes to copy, is negative, you
    // cannot complete the copy.
    if ((sourceOffset < 0) || (targetOffset < 0) || (count < 0))
    {
        throw new System.ArgumentException("offset or bytes to copy is
negative");
    }

    // If the number of bytes from the offset to the end of the array is
    // less than the number of bytes you want to copy, you cannot complete
    // the copy.
    if ((source.Length - sourceOffset < count) ||
        (target.Length - targetOffset < count))
    {
        throw new System.ArgumentException("offset to end of array is less
than bytes to be copied");
    }

    // The following fixed statement pins the location of the source and
    // target objects in memory so that they will not be moved by garbage
    // collection.
    fixed (byte* pSource = source, pTarget = target)
    {
        // Copy the specified number of bytes from source to target.
        for (int i = 0; i < count; i++)
        {
            pTarget[targetOffset + i] = pSource[sourceOffset + i];
        }
```

```
    }
}

static void UnsafeCopyArrays()
{
    // Create two arrays of the same length.
    int length = 100;
    byte[] byteArray1 = new byte[length];
    byte[] byteArray2 = new byte[length];

    // Fill byteArray1 with 0 - 99.
    for (int i = 0; i < length; ++i)
    {
        byteArray1[i] = (byte)i;
    }

    // Display the first 10 elements in byteArray1.
    System.Console.WriteLine("The first 10 elements of the original are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray1[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of byteArray1 to byteArray2.
    Copy(byteArray1, 0, byteArray2, 0, length);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");

    // Copy the contents of the last 10 elements of byteArray1 to the
    // beginning of byteArray2.
    // The offset specifies where the copying begins in the source array.
    int offset = length - 10;
    Copy(byteArray1, offset, byteArray2, 0, length - offset);

    // Display the first 10 elements in the copy, byteArray2.
    System.Console.WriteLine("The first 10 elements of the copy are:");
    for (int i = 0; i < 10; ++i)
    {
        System.Console.Write(byteArray2[i] + " ");
    }
    System.Console.WriteLine("\n");
    /* Output:
        The first 10 elements of the original are:
        0 1 2 3 4 5 6 7 8 9

        The first 10 elements of the copy are:
        0 1 2 3 4 5 6 7 8 9
```

```
        The first 10 elements of the copy are:
        90 91 92 93 94 95 96 97 98 99
    */
}
```

Function pointers

C# provides delegate types to define safe function pointer objects. Invoking a delegate involves instantiating a type derived from System.Delegate and making a virtual method call to its `Invoke` method. This virtual call uses the `callvirt` IL instruction. In performance critical code paths, using the `calli` IL instruction is more efficient.

You can define a function pointer using the `delegate*` syntax. The compiler calls the function using the `calli` instruction rather than instantiating a `delegate` object and calling `Invoke`. The following code declares two methods that use a `delegate` or a `delegate*` to combine two objects of the same type. The first method uses a System.Func<T1,T2,TResult> delegate type. The second method uses a `delegate*` declaration with the same parameters and return type:

C#

```
public static T Combine<T>(Func<T, T, T> combinator, T left, T right) =>
    combinator(left, right);

public static unsafe T UnsafeCombine<T>(delegate*<T, T, T> combinator, T
left, T right) =>
    combinator(left, right);
```

The following code shows how you would declare a static local function and invoke the `UnsafeCombine` method using a pointer to that local function:

C#

```
int product = 0;
unsafe
{
    static int localMultiply(int x, int y) => x * y;
    product = UnsafeCombine(&localMultiply, 3, 4);
}
```

The preceding code illustrates several of the rules on the function accessed as a function pointer:

- Function pointers can only be declared in an `unsafe` context.

- Methods that take a `delegate*` (or return a `delegate*`) can only be called in an `unsafe` context.
- The `&` operator to obtain the address of a function is allowed only on `static` functions. (This rule applies to both member functions and local functions).

The syntax has parallels with declaring `delegate` types and using pointers. The `*` suffix on `delegate` indicates the declaration is a *function pointer*. The `&` when assigning a method group to a function pointer indicates the operation takes the address of the method.

You can specify the calling convention for a `delegate*` using the keywords `managed` and `unmanaged`. In addition, for `unmanaged` function pointers, you can specify the calling convention. The following declarations show examples of each. The first declaration uses the `managed` calling convention, which is the default. The next four use an `unmanaged` calling convention. Each specifies one of the ECMA 335 calling conventions: `Cdecl`, `Stdcall`, `Fastcall`, or `Thiscall`. The last declaration uses the `unmanaged` calling convention, instructing the CLR to pick the default calling convention for the platform. The CLR chooses the calling convention at run time.

C#

```
public static unsafe T ManagedCombine<T>(delegate* managed<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static unsafe T CDeclCombine<T>(delegate* unmanaged[Cdecl]<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
public static unsafe T StdcallCombine<T>(delegate* unmanaged[Stdcall]<T, T,
T> combinator, T left, T right) =>
    combinator(left, right);
public static unsafe T FastcallCombine<T>(delegate* unmanaged[Fastcall]<T,
T, T> combinator, T left, T right) =>
    combinator(left, right);
public static unsafe T ThiscallCombine<T>(delegate* unmanaged[Thiscall]<T,
T, T> combinator, T left, T right) =>
    combinator(left, right);
public static unsafe T UnmanagedCombine<T>(delegate* unmanaged<T, T, T>
combinator, T left, T right) =>
    combinator(left, right);
```

You can learn more about function pointers in the Function pointer feature spec.

C# language specification

For more information, see the Unsafe code chapter of the C# language specification.

C# preprocessor directives

09/24/2025

Although the compiler doesn't have a separate preprocessor, the directives described in this section are processed as if there were one. You use them to help in conditional compilation. Unlike C and C++ directives, you can't use these directives to create macros. A preprocessor directive must be the only instruction on a line.

File-based apps

File-based apps are programs that are compiled and run using `dotnet run Program.cs` (or any `*.cs` file). The C# compiler ignores these preprocessor directives, but the build system parses them to produce the output. These directives generate warnings when encountered in a project-based compilation.

The C# compiler ignores any preprocessor directive that starts with `#:` or `#!`.

The `#!` preprocessor directive enables unix shells to directly execute a C# file using `dotnet run`. For example:

```
C#

#!/usr/local/share/dotnet/dotnet run
Console.WriteLine("Hello");
```

The preceding code snippet informs a Unix shell to execute the file using `/usr/local/share/dotnet/dotnet run`. (Your installation directory for the `dotnet` CLI can be different on different Unix or macOS distributions). The `#!` line must be the first line in the file, and the following tokens are the program to run. You need to enable the *execute* (`x`) permission on the C# file for that feature.

The `#:` directives that are used in file-based apps include:

- `#:sdk`:

  The first instance specifies the value for the `<Project Sdk="value" />` node. Subsequent instances specify the `<Sdk Name="value" Version="version" />` node. The version can be omitted (i.e. if specified in global.json or included in .NET SDK). For example:

  ```
  C#
  ```

```
#:sdk Microsoft.NET.Sdk.Web
#:sdk Aspire.AppHost.Sdk@9.4.1
```

The two preceding preprocessors is translated into:

```xml
<Project Sdk="Microsoft.NET.Sdk.Web" />
    <Sdk Name="Aspire.AppHost.Sdk" Version="9.4.1" />
```

- `#:property`:

  Instances of `#:property` are translated into property elements in a `<PropertyGroup>`. A token of the form `Name=value` must follow the `property` token. The following example directives are valid `property` tokens:

  ```
  #:property TargetFramework=net11.0
  #:property LangVersion=preview
  ```

  The preceding two properties are translated into:

  ```xml
  <TargetFramework>net11.0</TargetFramework>
  <LangVersion>preview</LangVersion>
  ```

- `#:package`:

  Instances of `#:package` are translated into `PackageReference` elements to include NuGet packages with the specified version to your file. For example:

  ```
  #:package System.CommandLine@2.0.0-*
  ```

  The preceding preprocessor token is translated into:

  ```xml
  <PackageReference Include="System.CommandLine" Version="2.0.0-*">
  ```

- `#:project`:

  Instances of `#:project` are translated into `ProjectReference` elements to include the project with the specified path to the project. For example:

  ```
  C#

  #:project ../Path/To.Example
  ```

  The preceding preprocessor token is translated into:

  ```
  XML

  <ProjectReference Include="../Path/To.Example.csproj" />
  ```

Tools can add new tokens following the `#:` convention.

Nullable context

The `#nullable` preprocessor directive sets the *annotations* and *warning* flags in the *nullable context*. This directive controls whether nullable annotations have effect, and whether nullability warnings are given. Each flag is either *disabled* or *enabled*.

Both contexts can be specified at the project level (outside of C# source code) adding the Nullable element to the `PropertyGroup` element. The `#nullable` directive controls the annotation and warning flags and takes precedence over the project-level settings. A directive sets the flag it controls until another directive overrides it, or until the end of the source file.

The effect of the directives is as follows:

- `#nullable disable`: Sets the nullable context *disabled*.
- `#nullable enable`: Sets the nullable context *enabled*.
- `#nullable restore`: Restores the nullable context to project settings.
- `#nullable disable annotations`: Sets the annotations flag in the nullable context to *disabled*.
- `#nullable enable annotations`: Sets the annotations flag in the nullable context to *enabled*.
- `#nullable restore annotations`: Restores the annotations flag in the nullable context to project settings.
- `#nullable disable warnings`: Sets the warning flag in the nullable context to *disabled*.
- `#nullable enable warnings`: Sets the warning flag in the nullable context to *enabled*.

- `#nullable restore warnings`: Restores the warning flag in the nullable context to project settings.

Conditional compilation

You use four preprocessor directives to control conditional compilation:

- `#if`: Opens a conditional compilation, where code is compiled only if the specified symbol is defined.
- `#elif`: Closes the preceding conditional compilation and opens a new conditional compilation based on if the specified symbol is defined.
- `#else`: Closes the preceding conditional compilation and opens a new conditional compilation if the previous specified symbol isn't defined.
- `#endif`: Closes the preceding conditional compilation.

The build system is also aware of predefined preprocessor symbols representing different target frameworks in SDK-style projects. They're useful when creating applications that can target more than one .NET version.

Expand table

Target Frameworks	Symbols	Additional symbols (available in .NET 5+ SDKs)	Platform symbols (available only when you specify an OS-specific TFM)
.NET Framework	`NETFRAMEWORK`, `NET481`, `NET48`, `NET472`, `NET471`, `NET47`, `NET462`, `NET461`, `NET46`, `NET452`, `NET451`, `NET45`, `NET40`, `NET35`, `NET20`	`NET48_OR_GREATER`, `NET472_OR_GREATER`, `NET471_OR_GREATER`, `NET47_OR_GREATER`, `NET462_OR_GREATER`, `NET461_OR_GREATER`, `NET46_OR_GREATER`, `NET452_OR_GREATER`, `NET451_OR_GREATER`, `NET45_OR_GREATER`, `NET40_OR_GREATER`, `NET35_OR_GREATER`, `NET20_OR_GREATER`	
.NET Standard	`NETSTANDARD`, `NETSTANDARD2_1`, `NETSTANDARD2_0`, `NETSTANDARD1_6`, `NETSTANDARD1_5`, `NETSTANDARD1_4`, `NETSTANDARD1_3`, `NETSTANDARD1_2`,	`NETSTANDARD2_1_OR_GREATER`, `NETSTANDARD2_0_OR_GREATER`, `NETSTANDARD1_6_OR_GREATER`, `NETSTANDARD1_5_OR_GREATER`, `NETSTANDARD1_4_OR_GREATER`, `NETSTANDARD1_3_OR_GREATER`, `NETSTANDARD1_2_OR_GREATER`, `NETSTANDARD1_1_OR_GREATER`, `NETSTANDARD1_0_OR_GREATER`	

Target Frameworks	Symbols	Additional symbols (available in .NET 5+ SDKs)	Platform symbols (available only when you specify an OS-specific TFM)
	`NETSTANDARD1_1`, `NETSTANDARD1_0`		
.NET 5+ (and .NET Core)	`NET`, `NET9_0`, `NET8_0`, `NET7_0`, `NET6_0`, `NET5_0`, `NETCOREAPP`, `NETCOREAPP3_1`, `NETCOREAPP3_0`, `NETCOREAPP2_2`, `NETCOREAPP2_1`, `NETCOREAPP2_0`, `NETCOREAPP1_1`, `NETCOREAPP1_0`	`NET9_0_OR_GREATER`, `NET8_0_OR_GREATER`, `NET7_0_OR_GREATER`, `NET6_0_OR_GREATER`, `NET5_0_OR_GREATER`, `NETCOREAPP3_1_OR_GREATER`, `NETCOREAPP3_0_OR_GREATER`, `NETCOREAPP2_2_OR_GREATER`, `NETCOREAPP2_1_OR_GREATER`, `NETCOREAPP2_0_OR_GREATER`, `NETCOREAPP1_1_OR_GREATER`, `NETCOREAPP1_0_OR_GREATER`	`ANDROID`, `BROWSER`, `IOS`, `MACCATALYST`, `MACOS`, `TVOS`, `WINDOWS`, `[OS][version]` (for example `IOS15_1`), `[OS][version]_OR_GREATER` (for example `IOS15_1_OR_GREATER`)

> **Note**
>
> - Versionless symbols are defined regardless of the version you're targeting.
> - Version-specific symbols are only defined for the version you're targeting.
> - The `<framework>_OR_GREATER` symbols are defined for the version you're targeting and all earlier versions. For example, if you're targeting .NET Framework 2.0, the following symbols are defined: `NET20`, `NET20_OR_GREATER`, `NET11_OR_GREATER`, and `NET10_OR_GREATER`.
> - The `NETSTANDARD<x>_<y>_OR_GREATER` symbols are only defined for .NET Standard targets, and not for targets that implement .NET Standard, such as .NET Core and .NET Framework.
> - These are different from the target framework monikers (TFMs) used by the MSBuild TargetFramework property and NuGet.

> **Note**
>
> For traditional, non-SDK-style projects, you have to manually configure the conditional compilation symbols for the different target frameworks in Visual Studio via the project's properties pages.

Other predefined symbols include the `DEBUG` and `TRACE` constants. You can override the values set for the project using `#define`. The DEBUG symbol, for example, is automatically set depending on your build configuration properties ("Debug" or "Release" mode).

The C# compiler compiles the code between the `#if` directive and `#endif` directive only if the specified symbol is defined, or not defined when the `!` not operator is used. Unlike C and C++, a numeric value to a symbol can't be assigned. The `#if` statement in C# is Boolean and only tests whether the symbol is defined or not. For example, the following code is compiled when `DEBUG` is defined:

C#

```
#if DEBUG
    Console.WriteLine("Debug version");
#endif
```

The following code is compiled when `MYTEST` is **not** defined:

C#

```
#if !MYTEST
    Console.WriteLine("MYTEST is not defined");
#endif
```

You can use the operators == (equality) and != (inequality) to test for the bool values `true` or `false`. `true` means the symbol is defined. The statement `#if DEBUG` has the same meaning as `#if (DEBUG == true)`. You can use the && (and), || (or), and ! (not) operators to evaluate whether multiple symbols are defined. You can also group symbols and operators with parentheses.

The following example shows a complex directive that allows your code to take advantage of newer .NET features while remaining backward compatible. For example, imagine that you're using a NuGet package in your code, but the package only supports .NET 6 and up, as well as .NET Standard 2.0 and up:

C#

```
#if (NET6_0_OR_GREATER || NETSTANDARD2_0_OR_GREATER)
    Console.WriteLine("Using .NET 6+ or .NET Standard 2+ code.");
#else
    Console.WriteLine("Using older code that doesn't support the above .NET
versions.");
#endif
```

`#if`, along with the `#else`, `#elif`, `#endif`, `#define`, and `#undef` directives, lets you include or exclude code based on the existence of one or more symbols. Conditional compilation can be useful when compiling code for a debug build or when compiling for a specific configuration.

`#elif` lets you create a compound conditional directive. The `#elif` expression is evaluated if neither the preceding `#if` nor any preceding, optional, `#elif` directive expressions evaluate to `true`. If an `#elif` expression evaluates to `true`, the compiler evaluates all the code between the `#elif` and the next conditional directive. For example:

C#

```
#define VC7
//...
#if DEBUG
    Console.WriteLine("Debug build");
#elif VC7
    Console.WriteLine("Visual Studio 7");
#endif
```

`#else` lets you create a compound conditional directive, so that, if none of the expressions in the preceding `#if` or (optional) `#elif` directives evaluate to `true`, the compiler will evaluate all code between `#else` and the next `#endif`. `#endif`(#endif) must be the next preprocessor directive after `#else`.

`#endif` specifies the end of a conditional directive, which began with the `#if` directive.

The following example shows you how to define a `MYTEST` symbol on a file and then test the values of the `MYTEST` and `DEBUG` symbols. The output of this example depends on whether you built the project on **Debug** or **Release** configuration mode.

C#

```
#define MYTEST
using System;
public class MyClass
{
    static void Main()
    {
#if (DEBUG && !MYTEST)
        Console.WriteLine("DEBUG is defined");
#elif (!DEBUG && MYTEST)
        Console.WriteLine("MYTEST is defined");
#elif (DEBUG && MYTEST)
        Console.WriteLine("DEBUG and MYTEST are defined");
#else
        Console.WriteLine("DEBUG and MYTEST are not defined");
#endif
```

```
    }
}
```

The following example shows you how to test for different target frameworks so you can use newer APIs when possible:

C#

```
public class MyClass
{
    static void Main()
    {
#if NET40
        WebClient _client = new WebClient();
#else
        HttpClient _client = new HttpClient();
#endif
    }
    //...
}
```

Defining symbols

You use the following two preprocessor directives to define or undefine symbols for conditional compilation:

- `#define`: Define a symbol.
- `#undef`: Undefine a symbol.

You use `#define` to define a symbol. When you use the symbol as the expression passed to the `#if` directive, the expression evaluates to `true`, as the following example shows:

C#

```
#define VERBOSE

#if VERBOSE
   Console.WriteLine("Verbose output version");
#endif
```

> **Note**
>
> In C#, primitive constants should be defined using the const keyword. A `const` declaration creates a `static` member that can't be modified at runtime. The `#define` directive can't

be used to declare constant values as is typically done in C and C++. If you have several such constants, consider creating a separate "Constants" class to hold them.

Symbols can be used to specify conditions for compilation. You can test for the symbol with either `#if` or `#elif`. You can also use the ConditionalAttribute to perform conditional compilation. You can define a symbol, but you can't assign a value to a symbol. The `#define` directive must appear in the file before you use any instructions that aren't also preprocessor directives. You can also define a symbol with the **DefineConstants** compiler option. You can undefine a symbol with `#undef`.

Defining regions

You can define regions of code that can be collapsed in an outline using the following two preprocessor directives:

- `#region`: Start a region.
- `#endregion`: End a region.

`#region` lets you specify a block of code that you can expand or collapse when using the outlining feature of the code editor. In longer code files, it's convenient to collapse or hide one or more regions so that you can focus on the part of the file that you're currently working on. The following example shows how to define a region:

C#

```
#region MyClass definition
public class MyClass
{
    static void Main()
    {
    }
}
#endregion
```

A `#region` block must be terminated with an `#endregion` directive. A `#region` block can't overlap with an `#if` block. However, a `#region` block can be nested in an `#if` block, and an `#if` block can be nested in a `#region` block.

Error and warning information

You instruct the compiler to generate user-defined compiler errors and warnings, and control line information using the following directives:

- `#error`: Generate a compiler error with a specified message.
- `#warning`: Generate a compiler warning, with a specific message.
- `#line`: Change the line number printed with compiler messages.

`#error` lets you generate a CS1029 user-defined error from a specific location in your code. For example:

C#

```
#error Deprecated code in this method.
```

> **Note**
>
> The compiler treats `#error version` in a special way and reports a compiler error, CS8304, with a message containing the used compiler and language versions.

`#warning` lets you generate a CS1030 level one compiler warning from a specific location in your code. For example:

C#

```
#warning Deprecated code in this method.
```

`#line` lets you modify the compiler's line numbering and (optionally) the file name output for errors and warnings.

The following example shows how to report two warnings associated with line numbers. The `#line 200` directive forces the next line's number to be 200 (although the default is #6), and until the next `#line` directive, the filename will be reported as "Special." The `#line default` directive returns the line numbering to its default numbering, which counts the lines renumbered by the previous directive.

C#

```
class MainClass
{
    static void Main()
    {
#line 200 "Special"
        int i;
        int j;
#line default
        char c;
        float f;
```

```
#line hidden // numbering not affected
        string s;
        double d;
    }
}
```

Compilation produces the following output:

```
Console

Special(200,13): warning CS0168: The variable 'i' is declared but never used
Special(201,13): warning CS0168: The variable 'j' is declared but never used
MainClass.cs(9,14): warning CS0168: The variable 'c' is declared but never used
MainClass.cs(10,15): warning CS0168: The variable 'f' is declared but never used
MainClass.cs(12,16): warning CS0168: The variable 's' is declared but never used
MainClass.cs(13,16): warning CS0168: The variable 'd' is declared but never used
```

The `#line` directive might be used in an automated, intermediate step in the build process. For example, if lines were removed from the original source code file, but you still wanted the compiler to generate output based on the original line numbering in the file, you could remove lines and then simulate the original line numbering with `#line`.

The `#line hidden` directive hides the successive lines from the debugger, such that when the developer steps through the code, any lines between a `#line hidden` and the next `#line` directive (assuming that it isn't another `#line hidden` directive) will be stepped over. This option can also be used to allow ASP.NET to differentiate between user-defined and machine-generated code. Although ASP.NET is the primary consumer of this feature, it's likely that more source generators make use of it.

A `#line hidden` directive doesn't affect file names or line numbers in error reporting. That is, if the compiler finds an error in a hidden block, the compiler reports the current file name and line number of the error.

The `#line filename` directive specifies the file name you want to appear in the compiler output. By default, the actual name of the source code file is used. The file name must be in double quotation marks ("") and must follow a line number.

You can use a new form of the `#line` directive:

```
C#

#line (1, 1) - (5, 60) 10 "partial-class.cs"
/*34567*/int b = 0;
```

The components of this form are:

- `(1, 1)`: The start line and column for the first character on the line following the directive. In this example, the next line would be reported as line 1, column 1.
- `(5, 60)`: The end line and column for the marked region.
- `10`: The column offset for the `#line` directive to take effect. In this example, the 10th column would be reported as column one. The declaration `int b = 0;` begins at that column. This field is optional. If omitted, the directive takes effect on the first column.
- `"partial-class.cs"`: The name of the output file.

The preceding example would generate the following warning:

.NET CLI

```
partial-class.cs(1,5,1,6): warning CS0219: The variable 'b' is assigned but its
value is never used
```

After remapping, the variable, `b`, is on the first line, at character six, of the file `partial-class.cs`.

Domain-specific languages (DSLs) typically use this format to provide a better mapping from the source file to the generated C# output. The most common use of this extended `#line` directive is to remap warnings or errors that appear in a generated file to the original source. For example, consider this razor page:

razor

```
@page "/"
Time: @DateTime.NowAndThen
```

The property `DateTime.Now` was typed incorrectly as `DateTime.NowAndThen`. The generated C# for this razor snippet looks like the following, in `page.g.cs`:

C#

```
  _builder.Add("Time: ");
#line (2, 6) - (2, 27) 15 "page.razor"
  _builder.Add(DateTime.NowAndThen);
```

The compiler output for the preceding snippet is:

.NET CLI

```
page.razor(2, 2, 2, 27)error CS0117: 'DateTime' does not contain a definition for
'NowAndThen'
```

Line 2, column 6 in `page.razor` is where the text `@DateTime.NowAndThen` begins, noted by `(2, 6)` in the directive. That span of `@DateTime.NowAndThen` ends at line 2, column 27, noted by the `(2, 27)` in the directive. The text for `DateTime.NowAndThen` begins in column 15 of `page.g.cs`, noted by the `15` in the directive. The compiler reports the error in its location in `page.razor`. The developer can navigate directly to the error in their source code, not the generated source.

To see more examples of this format, see the feature specification in the section on examples.

Pragmas

`#pragma` gives the compiler special instructions for the compilation of the file in which it appears. The compiler must support the pragmas you use. In other words, you can't use `#pragma` to create custom preprocessing instructions.

- #pragma warning: Enable or disable warnings.
- #pragma checksum: Generate a checksum.

C#

```
#pragma pragma-name pragma-arguments
```

Where `pragma-name` is the name of a recognized pragma and `pragma-arguments` is the pragma-specific arguments.

#pragma warning

`#pragma warning` can enable or disable certain warnings. The `#pragma warning disable format` and `#pragma warning enable format` control how Visual Studio formats code blocks.

C#

```
#pragma warning disable warning-list
#pragma warning restore warning-list
```

Where `warning-list` is a comma-separated list of warning numbers, such as `414, CS3021`. The "CS" prefix is optional. When no warning numbers are specified, `disable` disables all warnings and `restore` enables all warnings.

> **Note**

To find warning numbers in Visual Studio, build your project and then look for the warning numbers in the **Output** window.

The `disable` takes effect beginning on the next line of the source file. The warning is restored on the line following the `restore`. If there's no `restore` in the file, the warnings are restored to their default state at the first line of any later files in the same compilation.

C#

```
// pragma_warning.cs
using System;

#pragma warning disable 414, CS3021
[CLSCompliant(false)]
public class C
{
    int i = 1;
    static void Main()
    {
    }
}
#pragma warning restore CS3021
[CLSCompliant(false)]  // CS3021
public class D
{
    int i = 1;
    public static void F()
    {
    }
}
```

Another form of the `warning` pragma disables or restores Visual Studio formatting commands in blocks of code:

C#

```
#pragma warning disable format
#pragma warning restore format
```

Visual Studio format commands don't modify text in blocks of code where `disable format` is in effect. Format commands, such as `Ctrl+K`, `Ctrl+D`, don't modify those regions of code. This pragma gives you fine control over the visual presentation of your code.

#pragma checksum

Generates checksums for source files to aid with debugging ASP.NET pages.

C#

```
#pragma checksum "filename" "{guid}" "checksum bytes"
```

Where `"filename"` is the name of the file that requires monitoring for changes or updates, `"{guid}"` is the Globally Unique Identifier (GUID) for the hash algorithm, and `"checksum_bytes"` is the string of hexadecimal digits representing the bytes of the checksum. Must be an even number of hexadecimal digits. An odd number of digits results in a compile-time warning, and the directive is ignored.

The Visual Studio debugger uses a checksum to make sure that it always finds the right source. The compiler computes the checksum for a source file, and then emits the output to the program database (PDB) file. The debugger then uses the PDB to compare against the checksum that it computes for the source file.

This solution doesn't work for ASP.NET projects, because the computed checksum is for the generated source file, rather than the .aspx file. To address this problem, `#pragma checksum` provides checksum support for ASP.NET pages.

When you create an ASP.NET project in Visual C#, the generated source file contains a checksum for the .aspx file, from which the source is generated. The compiler then writes this information into the PDB file.

If the compiler doesn't find a `#pragma checksum` directive in the file, it computes the checksum and writes the value to the PDB file.

C#

```
class TestClass
{
    static int Main()
    {
        #pragma checksum "file.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}"
"ab007f1d23d9" // New checksum
    }
}
```

C# compiler options

Article • 09/15/2021

This section describes the options interpreted by the C# compiler. Options are grouped into separate articles based on what they control, for example, language features, code generation, and output. Use the table of contents to navigate amongst them.

How to set options

There are two different ways to set compiler options in .NET projects:

- ***In your *.csproj file***

  You can add MSBuild properties for any compiler option in your **.csproj* file in XML format. The property name is the same as the compiler option. The value of the property sets the value of the compiler option. For example, the following project file snippet sets the `LangVersion` property.

  XML

  ```
  <PropertyGroup>
    <LangVersion>preview</LangVersion>
  </PropertyGroup>
  ```

  For more information on setting options in project files, see the article MSBuild properties for .NET SDK Projects.

- ***Using the Visual Studio Property pages***

  Visual Studio provides property pages to edit build properties. To learn more about them, see Manage project and solution properties - Windows or Manage project and solution properties - Mac.

.NET Framework projects

> **Important**
>
> This section applies to .NET Framework projects only.

In addition to the mechanisms described above, you can set compiler options using two additional methods for .NET Framework projects:

- **Command line arguments for .NET Framework projects**: .NET Framework projects use *csc.exe* instead of `dotnet build` to build projects. You can specify command line arguments to *csc.exe* for .NET Framework projects.
- **Compiled ASP.NET pages**: .NET Framework projects use a section of the *web.config* file for compiling pages. For the new build system, and ASP.NET Core projects, options are taken from the project file.

The word for some compiler options changed from *csc.exe* and .NET Framework projects to the new MSBuild system. The new syntax is used throughout this section. Both versions are listed at the top of each page. For *csc.exe*, any arguments are listed following the option and a colon. For example, the `-doc` option would be:

Console

```
-doc:DocFile.xml
```

You can invoke the C# compiler by typing the name of its executable file (*csc.exe*) at a command prompt.

For .NET Framework projects, you can also run *csc.exe* from the command line. Every compiler option is available in two forms: **-option** and **/option**. In .NET Framework web projects, you specify options for compiling code-behind in the *web.config* file. For more information, see <compiler> Element.

If you use the **Developer Command Prompt for Visual Studio** window, all the necessary environment variables are set for you. For information on how to access this tool, see Developer Command Prompt for Visual Studio.

The *csc.exe* executable file is usually located in the Microsoft.NET\Framework\<*Version*> folder under the *Windows* directory. Its location might vary depending on the exact configuration of a particular computer. If more than one version of .NET Framework is installed on your computer, you'll find multiple versions of this file. For more information about such installations, see How to: determine which versions of the .NET Framework are installed.

C# Compiler Options for language feature rules

Article • 09/27/2024

The following options control how the compiler interprets language features. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **CheckForOverflowUnderflow** / `-checked`: Generate overflow checks.
- **AllowUnsafeBlocks** / `-unsafe`: Allow 'unsafe' code.
- **DefineConstants** / `-define`: Define conditional compilation symbol(s).
- **LangVersion** / `-langversion`: Specify language version such as `default` (latest major version), or `latest` (latest version, including minor versions).
- **Nullable** / `-nullable`: Enable nullable context, or nullable warnings.

> **Note**
>
> Refer to Compiler options for more information on configuring these options for your project.

CheckForOverflowUnderflow

The **CheckForOverflowUnderflow** option controls the default overflow-checking context that defines the program behavior if integer arithmetic overflows.

XML

```
<CheckForOverflowUnderflow>true</CheckForOverflowUnderflow>
```

When **CheckForOverflowUnderflow** is `true`, the default context is a checked context and overflow checking is enabled; otherwise, the default context is an unchecked context. The default value for this option is `false`, that is, overflow checking is disabled.

You can also explicitly control the overflow-checking context for the parts of your code by using the `checked` and `unchecked` statements.

For information about how the overflow-checking context affects operations and what operations are affected, see the article about checked and unchecked statements.

AllowUnsafeBlocks

The **AllowUnsafeBlocks** compiler option allows code that uses the unsafe keyword to compile. The default value for this option is `false`, meaning unsafe code isn't allowed.

```XML
<AllowUnsafeBlocks>true</AllowUnsafeBlocks>
```

For more information about unsafe code, see Unsafe Code and Pointers.

DefineConstants

The **DefineConstants** option defines symbols in all source code files of your program.

```XML
<DefineConstants>name;name2</DefineConstants>
```

This option specifies the names of one or more symbols that you want to define. The **DefineConstants** option has the same effect as the #define preprocessor directive except that the compiler option is in effect for all files in the project. A symbol remains defined in a source file until an #undef directive in the source file removes the definition. When you use the `-define` option, an `#undef` directive in one file has no effect on other source code files in the project. You can use symbols created by this option with #if, #else, #elif, and #endif to compile source files conditionally. The C# compiler itself defines no symbols or macros that you can use in your source code; all symbol definitions must be user-defined.

> **Note**
>
> The C# `#define` directive does not allow a symbol to be given a value, as in languages such as C++. For example, `#define` cannot be used to create a macro or to define a constant. If you need to define a constant, use an `enum` variable. If you want to create a C++ style macro, consider alternatives such as generics. Since macros are notoriously error-prone, C# disallows their use but provides safer alternatives.

LangVersion

The default language version for the C# compiler depends on the target framework for your application and the version of the SDK or Visual Studio installed. Those rules are defined in C# language versioning.

> **⚠ 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.

The **LangVersion** option causes the compiler to accept only syntax that is included in the specified C# language specification, for example:

XML

```xml
<LangVersion>9.0</LangVersion>
```

The following values are valid:

Expand table

Value	Meaning
`preview`	The compiler accepts all valid language syntax from the latest preview version.
`latest`	The compiler accepts syntax from the latest released version of the compiler (including minor version).
`latestMajor` or `default`	The compiler accepts syntax from the latest released major version of the compiler.
`14.0`	The compiler accepts only syntax that is included in C# 14 or lower.
`13.0`	The compiler accepts only syntax that is included in C# 13 or lower.
`12.0`	The compiler accepts only syntax that is included in C# 12 or lower.
`11.0`	The compiler accepts only syntax that is included in C# 11 or lower.
`10.0`	The compiler accepts only syntax that is included in C# 10 or lower.
`9.0`	The compiler accepts only syntax that is included in C# 9 or lower.
`8.0`	The compiler accepts only syntax that is included in C# 8.0 or lower.

Value	Meaning
`7.3`	The compiler accepts only syntax that is included in C# 7.3 or lower.
`7.2`	The compiler accepts only syntax that is included in C# 7.2 or lower.
`7.1`	The compiler accepts only syntax that is included in C# 7.1 or lower.
`7`	The compiler accepts only syntax that is included in C# 7.0 or lower.
`6`	The compiler accepts only syntax that is included in C# 6.0 or lower.
`5`	The compiler accepts only syntax that is included in C# 5.0 or lower.
`4`	The compiler accepts only syntax that is included in C# 4.0 or lower.
`3`	The compiler accepts only syntax that is included in C# 3.0 or lower.
`ISO-2` or `2`	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
`ISO-1` or `1`	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

Considerations

- To ensure that your project uses the default compiler version recommended for your target framework, don't use the **LangVersion** option. You can update the target framework to access newer language features.
- Specifying **LangVersion** with the `default` value is different from omitting the **LangVersion** option. Specifying `default` uses the latest version of the language that the compiler supports, without taking into account the target framework. For example, building a project that targets .NET 6 from Visual Studio version 17.6 uses C# 10 if **LangVersion** isn't specified, but uses C# 11 if **LangVersion** is set to `default`.
- Metadata referenced by your C# application isn't subject to the **LangVersion** compiler option.
- Because each version of the C# compiler contains extensions to the language specification, **LangVersion** doesn't give you the equivalent functionality of an earlier version of the compiler.
- While C# version updates generally coincide with major .NET releases, the new syntax and features aren't necessarily tied to that specific framework version. Each

specific feature has its own minimum .NET API or common language runtime requirements that may allow it to run on downlevel frameworks by including NuGet packages or other libraries.

- Regardless of which **LangVersion** setting you use, use the current version of the common language runtime to create your *.exe* or *.dll*. One exception is friend assemblies and ModuleAssemblyName, which work under **-langversion:ISO-1**.

For other ways to specify the C# language version, see C# language versioning.

For information about how to set this compiler option programmatically, see LanguageVersion.

C# language specification

Expand table

Version	Link	Description
C# 8.0 and later	download PDF	C# Language Specification Version 7: .NET Foundation
C# 7.3	download PDF	Standard ECMA-334 7th Edition
C# 6.0	download PDF	Standard ECMA-334 6th Edition
C# 5.0	Download PDF	Standard ECMA-334 5th Edition
C# 3.0	Download DOC	C# Language Specification Version 3.0: Microsoft Corporation
C# 2.0	Download PDF	Standard ECMA-334 4th Edition
C# 1.2	Download DOC	Standard ECMA-334 2nd Edition
C# 1.0	Download DOC	Standard ECMA-334 1st Edition

Minimum SDK version needed to support all language features

The following table lists the minimum versions of the SDK with the C# compiler that supports the corresponding language version:

Expand table

C# version	Minimum SDK version
C# 12	Microsoft Visual Studio/Build Tools 2022 version 17.8, or .NET 8 SDK
C# 11	Microsoft Visual Studio/Build Tools 2022 version 17.4, or .NET 7 SDK
C# 10	Microsoft Visual Studio/Build Tools 2022, or .NET 6 SDK
C# 9.0	Microsoft Visual Studio/Build Tools 2019 version 16.8, or .NET 5 SDK
C# 8.0	Microsoft Visual Studio/Build Tools 2019, version 16.3, or .NET Core 3.0 SDK
C# 7.3	Microsoft Visual Studio/Build Tools 2017, version 15.7
C# 7.2	Microsoft Visual Studio/Build Tools 2017, version 15.5
C# 7.1	Microsoft Visual Studio/Build Tools 2017, version 15.3
C# 7.0	Microsoft Visual Studio/Build Tools 2017
C# 6	Microsoft Visual Studio/Build Tools 2015
C# 5	Microsoft Visual Studio/Build Tools 2012 or bundled .NET Framework 4.5 compiler
C# 4	Microsoft Visual Studio/Build Tools 2010 or bundled .NET Framework 4.0 compiler
C# 3	Microsoft Visual Studio/Build Tools 2008 or bundled .NET Framework 3.5 compiler
C# 2	Microsoft Visual Studio/Build Tools 2005 or bundled .NET Framework 2.0 compiler
C# 1.0/1.2	Microsoft Visual Studio/Build Tools .NET 2002 or bundled .NET Framework 1.0 compiler

Nullable

The **Nullable** option lets you specify the nullable context. It can be set in the project's configuration using the `<Nullable>` tag:

XML

```
<Nullable>enable</Nullable>
```

The argument must be one of `enable`, `disable`, `warnings`, or `annotations`. The `enable` argument enables the nullable context. Specifying `disable` will disable the nullable context. When you specify the `warnings` argument, the nullable warning context is enabled. When you specify the `annotations` argument, the nullable annotation context is enabled. The values are described and explained in the article on Nullable contexts.

You can learn more about the tasks involved in enabling nullable reference types in an existing codebase in our article on nullable migration strategies.

> **Note**
>
> When there's no value set, the default value `disable` is applied, however the .NET 6 templates are by default provided with the **Nullable** value set to `enable`.

Flow analysis is used to infer the nullability of variables within executable code. The inferred nullability of a variable is independent of the variable's declared nullability. Method calls are analyzed even when they're conditionally omitted. For instance, Debug.Assert in release mode.

Invocation of methods annotated with the following attributes will also affect flow analysis:

- Simple pre-conditions: AllowNullAttribute and DisallowNullAttribute
- Simple post-conditions: MaybeNullAttribute and NotNullAttribute
- Conditional post-conditions: MaybeNullWhenAttribute and NotNullWhenAttribute
- DoesNotReturnIfAttribute (for example, `DoesNotReturnIf(false)` for Debug.Assert) and DoesNotReturnAttribute
- NotNullIfNotNullAttribute
- Member post-conditions: MemberNotNullAttribute(String) and MemberNotNullAttribute(String[])

> **Important**
>
> The global nullable context does not apply for generated code files. Regardless of this setting, the nullable context is *disabled* for any source file marked as generated. 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.

C# Compiler Options that control compiler output

Article • 07/12/2022

The following options control compiler output generation.

Expand table

MSBuild	*csc.exe*	Description
DocumentationFile	`-doc:`	Generate XML doc file from `///` comments.
OutputAssembly	`-out:`	Specify the output assembly file.
PlatformTarget	`-platform:`	Specify the target platform CPU.
ProduceReferenceAssembly	`-refout:`	Generate a reference assembly.
TargetType	`-target:`	Specify the type of the output assembly.

> **Note**
>
> Refer to Compiler options for more information on configuring these options for your project.

DocumentationFile

The **DocumentationFile** option allows you to place documentation comments in an XML file. To learn more about documenting your code, see Recommended Tags for Documentation Comments. The value specifies the path to the output XML file. The XML file contains the comments in the source code files of the compilation.

XML

```
<DocumentationFile>path/to/file.xml</DocumentationFile>
```

The source code file that contains Main or top-level statements is output first into the XML. You'll often want to use the generated .xml file with IntelliSense. The *.xml* filename must be the same as the assembly name. The *.xml* file must be in the same directory as the assembly. When the assembly is referenced in a Visual Studio project, the *.xml* file is found as well. For more information about generating code comments, see Supplying

Code Comments. Unless you compile with <TargetType:Module>, `file` will contain `<assembly>` and `</assembly>` tags specifying the name of the file containing the assembly manifest for the output file. For examples, see How to use the XML documentation features.

> **Note**
>
> The **DocumentationFile** option applies to all files in the project. To disable warnings related to documentation comments for a specific file or section of code, use #pragma warning.

This option can be used in any .NET SDK-style project. For more information, see DocumentationFile property.

OutputAssembly

The **OutputAssembly** option specifies the name of the output file. The output path specifies the folder where compiler output is placed.

XML

```
<OutputAssembly>folder</OutputAssembly>
```

Specify the full name and extension of the file you want to create. If you don't specify the name of the output file, MSBuild uses the name of the project to specify the name of the output assembly. Old style projects use the following rules:

- An .exe will take its name from the source code file that contains the `Main` method or top-level statements.
- A .dll or .netmodule will take its name from the first source code file.

Any modules produced as part of a compilation become files associated with any assembly also produced in the compilation. Use ildasm.exe to view the assembly manifest to see the associated files.

The **OutputAssembly** compiler option is required in order for an exe to be the target of a friend assembly.

PlatformTarget

Specifies which version of the CLR can run the assembly.

XML

```
<PlatformTarget>anycpu</PlatformTarget>
```

- **anycpu** (default) compiles your assembly to run on any platform. Your application runs as a 64-bit process whenever possible and falls back to 32-bit when only that mode is available.
- **anycpu32bitpreferred** compiles your assembly to run on any platform. Your application runs in 32-bit mode on systems that support both 64-bit and 32-bit applications. You can specify this option only for projects that target .NET Framework 4.5 or later.
- **ARM** compiles your assembly to run on a computer that has an Advanced RISC Machine (ARM) processor.
- **ARM64** compiles your assembly to run by the 64-bit CLR on a computer that has an Advanced RISC Machine (ARM) processor that supports the A64 instruction set.
- **x64** compiles your assembly to be run by the 64-bit CLR on a computer that supports the AMD64 or EM64T instruction set.
- **x86** compiles your assembly to be run by the 32-bit, x86-compatible CLR.
- **Itanium** compiles your assembly to be run by the 64-bit CLR on a computer with an Itanium processor.

On a 64-bit Windows operating system:

- Assemblies compiled with **x86** execute on the 32-bit CLR running under WOW64.
- A DLL compiled with the **anycpu** executes on the same CLR as the process into which it's loaded.
- Executables that are compiled with the **anycpu** execute on the 64-bit CLR.
- Executables compiled with **anycpu32bitpreferred** execute on the 32-bit CLR.

The **anycpu32bitpreferred** setting is valid only for executable (.EXE) files, and it requires .NET Framework 4.5 or later. For more information about developing an application to run on a Windows 64-bit operating system, see 64-bit Applications.

You set the **PlatformTarget** option from **Build** properties page for your project in Visual Studio.

The behavior of **anycpu** has some additional nuances on .NET Core and .NET 5 and later releases. When you set **anycpu**, publish your app and execute it with either the x86 `dotnet.exe` or the x64 `dotnet.exe`. For self-contained apps, the `dotnet publish` step packages the executable for the configure RID.

ProduceReferenceAssembly

The **ProduceReferenceAssembly** option controls whether the compiler produces reference assemblies.

```xml
<ProduceReferenceAssembly>true</ProduceReferenceAssembly>
```

Reference assemblies are a special type of assembly that contain only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools. Reference assemblies exclude all member implementations and declarations of private members. Those members have no observable impact on their API contract. For more information, see Reference assemblies in the .NET Guide.

The **ProduceReferenceAssembly** and **ProduceOnlyReferenceAssembly** options are mutually exclusive.

You generally don't need to work directly with reference assembly files. By default, reference assemblies are generated in a `ref` subfolder of the intermediate path (i.e. `obj/ref/`). To generate them under the output directory instead (i.e. `bin/ref/`) set `ProduceReferenceAssemblyInOutDir` to `true` in your project.

.NET SDK 6.0.200 made a change that moved reference assemblies from the output directory to the intermediate directory by default.

TargetType

The **TargetType** compiler option can be specified in one of the following forms:

- **library**: to create a code library. **library** is the default value.
- **exe**: to create an .exe file.
- **module** to create a module.
- **winexe** to create a Windows program.
- **winmdobj** to create an intermediate *.winmdobj* file.
- **appcontainerexe** to create an .exe file for Windows 8.x Store apps.

> ⓘ Note
>
> For .NET Framework targets, unless you specify **module**, this option causes a .NET Framework assembly manifest to be placed in an output file. For more information, see Assemblies in .NET and Common Attributes.

XML

```
<TargetType>library</TargetType>
```

The compiler creates only one assembly manifest per compilation. Information about all files in a compilation is placed in the assembly manifest. When producing multiple output files at the command line, only one assembly manifest can be created and it must go into the first output file specified on the command line.

If you create an assembly, you can indicate that all or part of your code is CLS-compliant with the CLSCompliantAttribute attribute.

library

The **library** option causes the compiler to create a dynamic-link library (DLL) rather than an executable file (EXE). The DLL will be created with the *.dll* extension. Unless otherwise specified with the **OutputAssembly** option, the output file name takes the name of the first input file. When building a *.dll* file, a Main method isn't required.

exe

The **exe** option causes the compiler to create an executable (EXE), console application. The executable file will be created with the .exe extension. Use **winexe** to create a Windows program executable. Unless otherwise specified with the **OutputAssembly** option, the output file name takes the name of the input file that contains the entry point (Main method or top-level statements). One and only one entry point is required in the source code files that are compiled into an .exe file. The **StartupObject** compiler option lets you specify which class contains the `Main` method, in cases where your code has more than one class with a `Main` method.

module

This option causes the compiler to not generate an assembly manifest. By default, the output file created by compiling with this option will have an extension of *.netmodule*. A file that doesn't have an assembly manifest cannot be loaded by the .NET runtime. However, such a file can be incorporated into the assembly manifest of an assembly with **AddModules**. If more than one module is created in a single compilation, internal types in one module will be available to other modules in the compilation. When code in one module references `internal` types in another module, then both modules must be incorporated into an assembly manifest, with **AddModules**. Creating a module isn't supported in the Visual Studio development environment.

winexe

The **winexe** option causes the compiler to create an executable (EXE), Windows program. The executable file will be created with the .exe extension. A Windows program is one that provides a user interface from either the .NET library or with the Windows APIs. Use **exe** to create a console application. Unless otherwise specified with the **OutputAssembly** option, the output file name takes the name of the input file that contains the Main method. One and only one `Main` method is required in the source code files that are compiled into an .exe file. The **StartupObject** option lets you specify which class contains the `Main` method, in cases where your code has more than one class with a `Main` method.

winmdobj

If you use the **winmdobj** option, the compiler creates an intermediate *.winmdobj* file that you can convert to a Windows Runtime binary (*.winmd*) file. The *.winmd* file can then be consumed by JavaScript and C++ programs, in addition to managed language programs.

The **winmdobj** setting signals to the compiler that an intermediate module is required. The *.winmdobj* file can then be fed through the WinMDExp export tool to produce a Windows metadata (*.winmd*) file. The *.winmd* file contains both the code from the original library and the WinMD metadata that is used by JavaScript or C++ and by the Windows Runtime. The output of a file that's compiled by using the **winmdobj** compiler option is used only as input for the WimMDExp export tool. The *.winmdobj* file itself isn't referenced directly. Unless you use the **OutputAssembly** option, the output file name takes the name of the first input file. A Main method isn't required.

appcontainerexe

If you use the **appcontainerexe** compiler option, the compiler creates a Windows executable (*.exe*) file that must be run in an app container. This option is equivalent to -target:winexe but is designed for Windows 8.x Store apps.

To require the app to run in an app container, this option sets a bit in the Portable Executable (PE) file. When that bit is set, an error occurs if the CreateProcess method tries to launch the executable file outside an app container. Unless you use the **OutputAssembly** option, the output file name takes the name of the input file that contains the Main method.

C# Compiler Options that specify inputs

Article • 09/15/2021

The following options control compiler inputs. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **References** / `-reference` or `-references`: Reference metadata from the specified assembly file or files.
- **AddModules** / `-addmodule`: Add a module (created with `target:module` to this assembly.)
- **EmbedInteropTypes** / `-link`: Embed metadata from the specified interop assembly files.

References

The **References** option causes the compiler to import public type information in the specified file into the current project, enabling you to reference metadata from the specified assembly files.

XML

```
<Reference Include="filename" />
```

`filename` is the name of a file that contains an assembly manifest. To import more than one file, include a separate **Reference** element for each file. You can define an alias as a child element of the **Reference** element:

XML

```
<Reference Include="filename.dll">
  <Aliases>LS</Aliases>
</Reference>
```

In the previous example, `LS` is the valid C# identifier that represents a root namespace that will contain all namespaces in the assembly *filename.dll*. The files you import must contain a manifest. Use **AdditionalLibPaths** to specify the directory in which one or more of your assembly references is located. The **AdditionalLibPaths** topic also discusses the directories in which the compiler searches for assemblies. In order for the compiler to recognize a type in an assembly, and not in a module, it needs to be forced to resolve the type, which you can do by defining an instance of the type. There are

other ways to resolve type names in an assembly for the compiler: for example, if you inherit from a type in an assembly, the type name will then be recognized by the compiler. Sometimes it is necessary to reference two different versions of the same component from within one assembly. To do this, use the **Aliases** element on the **References** element for each file to distinguish between the two files. This alias will be used as a qualifier for the component name, and will resolve to the component in one of the files.

> ⓘ **Note**
>
> In Visual Studio, use the **Add Reference** command. For more information, see How to: Add or Remove References By Using the Reference Manager.

AddModules

This option adds a module that was created with the `<TargetType>module</TargetType>` switch to the current compilation:

XML

```
<AddModule Include=file1 />
<AddModule Include=file2 />
```

Where `file`, `file2` are output files that contain metadata. The file can't contain an assembly manifest. To import more than one file, separate file names with either a comma or a semicolon. All modules added with **AddModules** must be in the same directory as the output file at run time. That is, you can specify a module in any directory at compile time but the module must be in the application directory at run time. If the module isn't in the application directory at run time, you'll get a TypeLoadException. `file` can't contain an assembly. For example, if the output file was created with TargetType option of **module**, its metadata can be imported with **AddModules**.

If the output file was created with a TargetType option other than **module**, its metadata cannot be imported with **AddModules** but can be imported with the References option.

EmbedInteropTypes

Causes the compiler to make COM type information in the specified assemblies available to the project that you are currently compiling.

XML

```
<References>
  <EmbedInteropTypes>file1;file2;file3</EmbedInteropTypes>
</References>
```

Where `file1;file2;file3` is a semicolon-delimited list of assembly file names. If the file name contains a space, enclose the name in quotation marks. The **EmbedInteropTypes** option enables you to deploy an application that has embedded type information. The application can then use types in a runtime assembly that implement the embedded type information without requiring a reference to the runtime assembly. If various versions of the runtime assembly are published, the application that contains the embedded type information can work with the various versions without having to be recompiled. For an example, see Walkthrough: Embedding Types from Managed Assemblies.

Using the **EmbedInteropTypes** option is especially useful when you're working with COM interop. You can embed COM types so that your application no longer requires a primary interop assembly (PIA) on the target computer. The **EmbedInteropTypes** option instructs the compiler to embed the COM type information from the referenced interop assembly into the resulting compiled code. The COM type is identified by the CLSID (GUID) value. As a result, your application can run on a target computer that has installed the same COM types with the same CLSID values. Applications that automate Microsoft Office are a good example. Because applications like Office usually keep the same CLSID value across different versions, your application can use the referenced COM types as long as .NET Framework 4 or later is installed on the target computer and your application uses methods, properties, or events that are included in the referenced COM types. The **EmbedInteropTypes** option embeds only interfaces, structures, and delegates. Embedding COM classes isn't supported.

> **Note**
>
> When you create an instance of an embedded COM type in your code, you must create the instance by using the appropriate interface. Attempting to create an instance of an embedded COM type by using the CoClass causes an error.

Like the References compiler option, the **EmbedInteropTypes** compiler option uses the Csc.rsp response file, which references frequently used .NET assemblies. Use the NoConfig compiler option if you don't want the compiler to use the Csc.rsp file.

C#

```
// The following code causes an error if ISampleInterface is an embedded
interop type.
ISampleInterface<SampleType> sample;
```

Types that have a generic parameter whose type is embedded from an interop assembly cannot be used if that type is from an external assembly. This restriction doesn't apply to interfaces. For example, consider the Range interface that is defined in the Microsoft.Office.Interop.Excel assembly. If a library embeds interop types from the Microsoft.Office.Interop.Excel assembly and exposes a method that returns a generic type that has a parameter whose type is the Range interface, that method must return a generic interface, as shown in the following code example.

C#

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using Microsoft.Office.Interop.Excel;

public class Utility
{
    // The following code causes an error when called by a client assembly.
    public List<Range> GetRange1()
    {
        return null;
    }

    // The following code is valid for calls from a client assembly.
    public IList<Range> GetRange2()
    {
        return null;
    }
}
```

In the following example, client code can call the method that returns the IList generic interface without error.

C#

```
public class Client
{
    public void Main()
    {
        Utility util = new Utility();

        // The following code causes an error.
        List<Range> rangeList1 = util.GetRange1();
```

```
        // The following code is valid.
        List<Range> rangeList2 = (List<Range>)util.GetRange2();
    }
}
```

C# Compiler Options to report errors and warnings

05/31/2025

The following options control how the compiler reports errors and warnings.

Expand table

MSBuild syntax	*csc.exe* syntax	Description
`WarningLevel`	`-warn`	Set warning level. More info.
`AnalysisLevel`	/	Set optional warning level. More info.
`TreatWarningsAsErrors`	`-warnaserror`	Treat all warnings as errors. More info.
`WarningsAsErrors`	`-warnaserror+`	Treat one or more warnings as errors. More info.
`WarningsNotAsErrors`	`-warnaserror-`	Treat one or more warnings not as errors. More info.
`NoWarn`	`-nowarn`	Set a list of disabled warnings. More info.
`CodeAnalysisRuleSet`	`-ruleset`	Specify a ruleset file that disables specific diagnostics. More info.
`ErrorLog`	`-errorlog`	Specify a file to log all compiler and analyzer diagnostics. More info.
`ReportAnalyzer`	`-reportanalyzer`	Report extra analyzer information, such as execution time. More info.

> **Note**
>
> For more information about configuring these options for your project, see the Compiler options. For more information on the available switches, see the MSBuild command line switches article in the Visual Studio documentation.

WarningLevel

The **WarningLevel** option specifies the warning level for the compiler to display.

```
XML
```

```
<WarningLevel>3</WarningLevel>
```

The element value is the warning level you want displayed for the compilation: Lower numbers show only high severity warnings. Higher numbers show more warnings. The value must be zero or a positive integer:

Expand table

Warning level	Meaning
0	Turns off emission of all warning messages.
1	Displays severe warning messages.
2	Displays level 1 warnings plus certain, less-severe warnings, such as warnings about hiding class members.
3	Displays level 2 warnings plus certain, less-severe warnings, such as warnings about expressions that always evaluate to `true` or `false`.
4 (default)	Displays all level 3 warnings plus informational warnings.

> ⚠ **Warning**
>
> The compiler command line accepts values greater than 4 to enable warning wave warnings.

To get information about an error or warning, you can look up the error code in the Help Index. For other ways to get information about an error or warning, see C# Compiler Errors. Use **TreatWarningsAsErrors** to treat all warnings as errors. Use **DisabledWarnings** to disable certain warnings.

Analysis level

The **AnalysisLevel** option specifies higher warning waves and analyzers to enable in .NET 7 and later projects. Warning wave warnings are extra checks that improve your code, or ensure it remains compatible with upcoming releases. Analyzers provide lint-like capability to improve your code.

Beginning with the .NET 7 SDK, **AnalysisLevel** and **WarningLevel** are set based on these rules:

- The default **AnalysisLevel** matches the Target Framework Moniker (TFM) from the project file.
- The default **WarningLevel** matches the value for **AnalysisLevel**.
- The default **WarningLevel** is 4 for .NET Framework projects.

Before the .NET 7 SDK, the **AnalysisLevel** overwrote the **WarningLevel**.

XML

```xml
<AnalysisLevel>preview</AnalysisLevel>
```

Expand table

Analysis level	Meaning
5	Displays all optional warning wave 5 warnings.
6	Displays all optional warning wave 6 warnings.
7	Displays all optional warning wave 7 warnings.
latest (default)	Displays all informational warnings up to and including the current release.
preview	Displays all informational warnings up to and including the latest preview release.
none	Turns off all informational warnings.

For more information on optional warnings, see Warning waves.

To get information about an error or warning, you can look up the error code in the Help Index. For other ways to get information about an error or warning, see C# Compiler Errors. Use **TreatWarningsAsErrors** to treat all warnings as errors. Use **NoWarn** to disable certain warnings.

TreatWarningsAsErrors

The **TreatWarningsAsErrors** option treats all warnings as errors. You can also use the **WarningsAsErrors** to set only some warnings as errors. If you turn on **TreatWarningsAsErrors**, you can use **WarningsNotAsErrors** to list warnings that shouldn't be treated as errors.

XML

```xml
<TreatWarningsAsErrors>true</TreatWarningsAsErrors>
```

All warning messages are instead reported as errors. The build process halts (no output files are built). By default, **TreatWarningsAsErrors** isn't in effect, which means warnings don't

prevent the generation of an output file. Optionally, if you want only a few specific warnings to be treated as errors, you can specify a comma-separated list of warning numbers to treat as errors. The set of all nullability warnings can be specified with the **Nullable** shorthand. Use **WarningLevel** to specify the level of warnings that you want the compiler to display. Use **NoWarn** to disable certain warnings.

> **ⓘ Important**
>
> There are two subtle differences between using the `<TreatWarningsAsErrors>` element in your *csproj* file, and using the `warnaserror` MSBuild command line switch. *TreatWarningsAsErrors* only impacts the C# compiler, not any other MSBuild tasks in your *csproj* file. The `warnaserror` command line switch impacts all tasks. Secondly, the compiler doesn't produce any output on any warnings when *TreatWarningsAsErrors* is used. The compiler produces output when the `warnaserror` command line switch is used.

WarningsAsErrors and WarningsNotAsErrors

The **WarningsAsErrors** and **WarningsNotAsErrors** options override the **TreatWarningsAsErrors** option for a list of warnings. This option can be used with all *CS* warnings. The "CS" prefix is optional. You can use either the number, or "CS" followed by the error or warning number. For other elements that affect warnings, see the Common MSBuild properties. In addition to the list of warning IDs, you can also specify the string `nullable`, which treats all warnings related to nullability as errors.

Enable warnings 0219, 0168, and all nullable warnings as errors:

XML

```
<WarningsAsErrors>0219;CS0168;nullable</WarningsAsErrors>
```

Disable the same warnings as errors:

XML

```
<WarningsNotAsErrors>0219;CS0168;nullable</WarningsNotAsErrors>
```

You use **WarningsAsErrors** to configure a set of warnings as errors. Use **WarningsNotAsErrors** to configure a set of warnings that shouldn't be errors when you set all warnings as errors.

NoWarn

The **NoWarn** option lets you suppress the compiler from displaying one or more warnings, where `warningnumber1`, `warningnumber2` are warning numbers that you want the compiler to suppress. Separate multiple warning numbers with a comma. You can specify `nullable` to disable all warnings related to nullability.

```
XML

<NoWarn>warningnumber1,warningnumber2</NoWarn>
```

You need to specify only the numeric part of the warning identifier. For example, if you want to suppress *CS0028*, you could specify `<NoWarn>28</NoWarn>`. The compiler silently ignores warning numbers passed to **NoWarn** that were valid in previous releases, but aren't generated by the current compiler. For example, *CS0679* was valid in the compiler in Visual Studio .NET 2002 but was removed later.

The following warnings can't be suppressed by the **NoWarn** option:

- Compiler Warning (level 1) CS2002
- Compiler Warning (level 1) CS2023
- Compiler Warning (level 1) CS2029

Warnings are intended to be an indication of a potential problem with your code, so you should understand the risks of disabling any particular warning. Use **NoWarn** only when you're certain that a warning is a false positive and can't possibly be a runtime bug.

You might want to use a more targeted approach to disabling warnings:

- Most compilers provide ways to disable warnings just for certain lines of code, so that you can still review the warnings if they occur elsewhere in the same project. To suppress a warning only in a specific part of the code in C#, use #pragma warning.
- If your goal is to see more concise and focused output in your build log, you might want to change the build log verbosity. For more information, see How to: View, save, and configure build log files.

To add warning numbers to any previously set value for **NoWarn** without overwriting it, reference `$(NoWarn)` as shown in the following example:

```
XML

    <NoWarn>$(NoWarn);newwarningnumber3;newwarningnumber4</NoWarn>
```

CodeAnalysisRuleSet

Specify a ruleset file that configures specific diagnostics.

```
XML

<CodeAnalysisRuleSet>MyConfiguration.ruleset</CodeAnalysisRuleSet>
```

Where `MyConfiguration.ruleset` is the path to the ruleset file. For more information on using rule sets, see the article in the Visual Studio documentation on Rule sets.

ErrorLog

Specify a file to log all compiler and analyzer diagnostics.

```
XML

<ErrorLog>compiler-diagnostics.sarif</ErrorLog>
```

The **ErrorLog** option causes the compiler to output a Static Analysis Results Interchange Format (SARIF) log. Tools that analyze compiler and analyzer results read SARIF logs.

You can specify the SARIF format using the `version` argument to the `ErrorLog` element:

```
XML

<ErrorLog>logVersion21.json,version=2.1</ErrorLog>
```

The separator can be either a comma (`,`) or a semicolon (`;`). Valid values for version are: "1", "2", and "2.1". The default is "1". "2" and "2.1" both mean SARIF version 2.1.0.

ReportAnalyzer

Report extra analyzer information, such as execution time.

```
XML

<ReportAnalyzer>true</ReportAnalyzer>
```

The **ReportAnalyzer** option causes the compiler to emit extra MSBuild log information that details the performance characteristics of analyzers in the build. It's typically used by analyzer authors as part of validating the analyzer.

> **ⓘ Important**
>
> The extra log information generated by this flag is only generated when the `-verbosity:detailed` command line option is used. For more information, see the **switches** article in the MSBuild documentation.

C# Compiler Options that control code generation

08/07/2025

The following options control code generation by the compiler. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **DebugType** / `-debug`: Emit (or don't emit) debugging information.
- **Optimize** / `-optimize`: Enable optimizations.
- **Deterministic** / `-deterministic`: Produce byte-for-byte equivalent output from the same input source.
- **ProduceOnlyReferenceAssembly** / `-refonly`: Produce a reference assembly, instead of a full assembly, as the primary output.

> **Note**
>
> Refer to Compiler options for more information on configuring these options for your project.

DebugType

The **DebugType** option causes the compiler to generate debugging information and place it in the output file or files. The default value is `portable` for both Debug and Release build configurations, which means PDB files are generated by default for all configurations.

XML

```
<DebugType>pdbonly</DebugType>
```

For all compiler versions starting with C# 6.0, there is no difference between *pdbonly* and *full*. Choose *pdbonly*. To change the location of the *.pdb* file, see **PdbFile**.

The following values are valid:

Expand table

Value	Meaning
`full`	Emit debugging information to *.pdb* file using default format for the current platform: **Windows**: A Windows pdb file.

Value	Meaning
	Linux/macOS: A Portable PDB file.
`pdbonly`	Same as `full`. See the note below for more information.
`portable`	Emit debugging information to .pdb file using cross-platform Portable PDB format.
`embedded`	Emit debugging information into the *.dll/.exe* itself (*.pdb* file is not produced) using Portable PDB format.
`none`	Don't produce a PDB file.

> **Important**
>
> The following information applies only to compilers older than C# 6.0. The value of this element can be either `full` or `pdbonly`. The *full* argument, which is in effect if you don't specify *pdbonly*, enables attaching a debugger to the running program. Specifying *pdbonly* allows source code debugging when the program is started in the debugger but will only display assembler when the running program is attached to the debugger. Use this option to create debug builds. If you use *Full*, be aware that there's some impact on the speed and size of JIT optimized code and a small impact on code quality with *full*. We recommend *pdbonly* or no PDB for generating release code. One difference between *pdbonly* and *full* is that with *full* the compiler emits a DebuggableAttribute, which is used to tell the JIT compiler that debug information is available. Therefore, you will get an error if your code contains the DebuggableAttribute set to false if you use *full*. For more information on how to configure the debug performance of an application, see Making an Image Easier to Debug.

Disable PDB generation for Release builds

To suppress PDB file generation for Release builds while keeping them for Debug builds, add the following property to your project file:

XML

```xml
<PropertyGroup Condition="'$(Configuration)' == 'Release'">
  <DebugType>none</DebugType>
</PropertyGroup>
```

Alternatively, you can set `DebugSymbols` to `false` for Release builds:

XML

```
<PropertyGroup Condition="'$(Configuration)' == 'Release'">
  <DebugSymbols>false</DebugSymbols>
</PropertyGroup>
```

> **Note**
>
> In .NET 8 and later versions, setting `DebugSymbols` to `false` should suppress PDB generation according to the breaking change documentation. However, the most reliable way to disable PDB generation is to explicitly set `DebugType` to `none`.

Optimize

The **Optimize** option enables or disables optimizations performed by the compiler to make your output file smaller, faster, and more efficient. The *Optimize* option is enabled by default for a *Release* build configuration. It is off by default for a *Debug* and any other build configuration.

XML

```
<Optimize>true</Optimize>
```

You set the **Optimize** option from **Build** properties page for your project in Visual Studio.

Optimize also tells the common language runtime to optimize code at run time. By default, optimizations are disabled. Specify **Optimize+** to enable optimizations. When building a module to be used by an assembly, use the same **Optimize** settings as used by the assembly. It's possible to combine the **Optimize** and **Debug** options.

Deterministic

Causes the compiler to produce an assembly whose byte-for-byte output is identical across compilations for identical inputs.

XML

```
<Deterministic>true</Deterministic>
```

For modern .NET projects, deterministic compilation is enabled by default (the `Deterministic` property defaults to `true`). When deterministic compilation is enabled, the timestamp and

MVID fields are replaced with values derived from a hash of all the compilation inputs, ensuring identical binary output for identical inputs.

When deterministic compilation is disabled (`<Deterministic>false</Deterministic>`), compiler output from a given set of inputs is unique, since the compiler adds a timestamp and an MVID (a Module.ModuleVersionId. Basically it is a GUID that uniquely identifies the module and version.) that is generated from random numbers.

You use the `<Deterministic>` option to produce a *deterministic assembly*, one whose binary content is identical across compilations as long as the input remains the same. The compiler considers the following inputs that affect determinism:

- The sequence of command-line parameters.
- The contents of the compiler's .rsp response file.
- The precise version of the compiler used, and its referenced assemblies.
- The current directory path.
- The binary contents of all files explicitly passed to the compiler either directly or indirectly, including:
  - Source files
  - Referenced assemblies
  - Referenced modules
  - Resources
  - The strong name key file
  - @ response files
  - Analyzers
  - Rulesets
  - Source Link data extracted from the repository (for example, git commit SHA, repository URL, etc.)
  - Other files that may be used by analyzers
- The current culture (for the language in which diagnostics and exception messages are produced).
- The default encoding (or the current code page) if the encoding isn't specified.
- The existence, non-existence, and contents of files on the compiler's search paths (specified, for example, by `-lib` or `-recurse`).
- The Common Language Runtime (CLR) platform on which the compiler is run.
- The value of `%LIBPATH%`, which can affect analyzer dependency loading.

Deterministic compilation can be used for establishing whether a binary is compiled from a trusted source. Deterministic output can be useful when the source is publicly available. It can also determine whether build steps that are dependent on changes to binary used in the build process.

ProduceOnlyReferenceAssembly

The **ProduceOnlyReferenceAssembly** option indicates that a reference assembly should be output instead of an implementation assembly, as the primary output. The **ProduceOnlyReferenceAssembly** parameter silently disables outputting PDBs, as reference assemblies cannot be executed.

XML

```
<ProduceOnlyReferenceAssembly>true</ProduceOnlyReferenceAssembly>
```

Reference assemblies are a special type of assembly. Reference assemblies contain only the minimum amount of metadata required to represent the library's public API surface. They include declarations for all members that are significant when referencing an assembly in build tools, but exclude all member implementations and declarations of private members that have no observable impact on their API contract. For more information, see Reference assemblies.

The **ProduceOnlyReferenceAssembly** and **ProduceReferenceAssembly** options are mutually exclusive.

C# compiler options for security

Article • 09/29/2022

The following options control compiler security options. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **PublicSign** / `-publicsign`: Publicly sign the assembly.
- **DelaySign** / `-delaysign`: Delay-sign the assembly using only the public portion of the strong name key.
- **KeyFile** / `-keyfile` : Specify a strong name key file.
- **KeyContainer** / `-keycontainer`: Specify a strong name key container.
- **HighEntropyVA** / `-highentropyva`: Enable high-entropy Address Space Layout Randomization (ASLR)

> ⓘ **Note**
>
> Refer to Compiler options for more information on configuring these options for your project.

PublicSign

This option causes the compiler to apply a public key but doesn't actually sign the assembly. The **PublicSign** option also sets a bit in the assembly that tells the runtime that the file is signed.

XML

```
<PublicSign>true</PublicSign>
```

The **PublicSign** option requires the use of the KeyFile or KeyContainer option. The **KeyFile** and **KeyContainer** options specify the public key. The **PublicSign** and **DelaySign** options are mutually exclusive. Sometimes called "fake sign" or "OSS sign", public signing includes the public key in an output assembly and sets the "signed" flag. Public signing doesn't actually sign the assembly with a private key. Developers use public sign for open-source projects. People build assemblies that are compatible with the released "fully signed" assemblies when they don't have access to the private key used to sign the assemblies. Since few consumers actually need to check if the assembly is fully signed, these publicly built assemblies are useable in almost every scenario where the fully signed one would be used.

DelaySign

This option causes the compiler to reserve space in the output file so that a digital signature can be added later.

```
XML

<DelaySign>true</DelaySign>
```

Use **DelaySign-** if you want a fully signed assembly. Use **DelaySign** if you only want to place the public key in the assembly. The **DelaySign** option has no effect unless used with **KeyFile** or **KeyContainer**. The **KeyContainer** and **PublicSign** options are mutually exclusive. When you request a fully signed assembly, the compiler hashes the file that contains the manifest (assembly metadata) and signs that hash with the private key. That operation creates a digital signature that is stored in the file that contains the manifest. When an assembly is delay signed, the compiler doesn't compute and store the signature. Instead, the compiler but reserves space in the file so the signature can be added later.

Using **DelaySign** allows a tester to put the assembly in the global cache. After testing, you can fully sign the assembly by placing the private key in the assembly using the Assembly Linker utility. For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

KeyFile

Specifies the filename containing the cryptographic key.

```
XML

<KeyFile>filename</KeyFile>
```

`file` is the name of the file containing the strong name key. When this option is used, the compiler inserts the public key from the specified file into the assembly manifest and then signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. If you compile with **-target:module**, the name of the key file is held in the module and incorporated into the assembly created when you compile an assembly with **AddModules**. You can also pass your encryption information to the compiler with **KeyContainer**. Use **DelaySign** if you want a partially signed assembly. In case both **KeyFile** and **KeyContainer** are specified in the same compilation, the compiler will first try the key container. If that succeeds, then the assembly is signed with the

information in the key container. If the compiler doesn't find the key container, it will try the file specified with **KeyFile**. If that succeeds, the assembly is signed with the information in the key file and the key information will be installed in the key container. On the next compilation, the key container will be valid. A key file might contain only the public key. For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

KeyContainer

Specifies the name of the cryptographic key container.

```
XML
```

```
<KeyContainer>container</KeyContainer>
```

`container` is the name of the strong name key container. When the **KeyContainer** option is used, the compiler creates a sharable component. The compiler inserts a public key from the specified container into the assembly manifest and signs the final assembly with the private key. To generate a key file, type `sn -k file` at the command line. `sn -i` installs the key pair into a container. This option isn't supported when the compiler runs on CoreCLR. To sign an assembly when building on CoreCLR, use the **KeyFile** option. If you compile with **TargetType**, the name of the key file is held in the module and incorporated into the assembly when you compile this module into an assembly with **AddModules**. You can also specify this option as a custom attribute (System.Reflection.AssemblyKeyNameAttribute) in the source code for any common intermediate language (CIL) module. You can also pass your encryption information to the compiler with **KeyFile**. Use **DelaySign** to add the public key to the assembly manifest but signing the assembly until it has been tested. For more information, see Creating and Using Strong-Named Assemblies and Delay Signing an Assembly.

HighEntropyVA

The **HighEntropyVA** compiler option tells the Windows kernel whether a particular executable supports high entropy Address Space Layout Randomization (ASLR).

```
XML
```

```
<HighEntropyVA>true</HighEntropyVA>
```

This option specifies that a 64-bit executable or an executable that is marked by the **PlatformTarget** compiler option supports a high entropy virtual address space. The option is enabled by default for all .NET Standard and .NET Core versions, and .NET Framework versions starting with .NET Framework 4.5.

The **HighEntropyVA** option enables compatible versions of the Windows kernel to use higher degrees of entropy when randomizing the address space layout of a process as part of ASLR. Using higher degrees of entropy means a larger number of addresses can be allocated to memory regions such as stacks and heaps. As a result, it's more difficult to guess the location of a particular memory region. The **HighEntropyVA** compiler option requires the target executable and any modules that it depends on can handle pointer values larger than 4 gigabytes (GB) when they're running as a 64-bit process.

C# Compiler Options that specify resources

Article • 09/15/2021

The following options control how the C# compiler creates or imports Win32 resources. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* syntax is shown in `code style`.

- **Win32Resource** / `-win32res`: Specify a Win32 resource file (.res).
- **Win32Icon** / `-win32icon`: Reference metadata from the specified assembly file or files.
- **Win32Manifest** / `-win32manifest`: Specify a Win32 manifest file (.xml).
- **NoWin32Manifest** / `-nowin32manifest`: Don't include the default Win32 manifest.
- **Resources** / `-resource`: Embed the specified resource (Short form: /res).
- **LinkResources** / `-linkresources`: Link the specified resource to this assembly.

> **Note**
>
> Refer to Compiler options for more information on configuring these options for your project.

Win32Resource

The **Win32Resource** option inserts a Win32 resource in the output file.

XML

```
<Win32Resource>filename</Win32Resource>
```

`filename` is the resource file that you want to add to your output file. A Win32 resource can contain version or bitmap (icon) information that would help identify your application in the File Explorer. If you don't specify this option, the compiler will generate version information based on the assembly version.

Win32Icon

The **Win32Icon** option inserts an .ico file in the output file, which gives the output file the desired appearance in the File Explorer.

XML

```
<Win32Icon>filename</Win32Icon>
```

`filename` is the *.ico* file that you want to add to your output file. An *.ico* file can be created with the Resource Compiler. The Resource Compiler is invoked when you compile a Visual C++ program; an *.ico* file is created from the *.rc* file.

Win32Manifest

Use the **Win32Manifest** option to specify a user-defined Win32 application manifest file to be embedded into a project's portable executable (PE) file.

XML

```
<Win32Manifest>filename</Win32Manifest>
```

`filename` is the name and location of the custom manifest file. By default, the C# compiler embeds an application manifest that specifies a requested execution level of "asInvoker". It creates the manifest in the same folder in which the executable is built. If you want to supply a custom manifest, for example to specify a requested execution level of "highestAvailable" or "requireAdministrator," use this option to specify the name of the file.

> ⓘ **Note**
>
> This option and the **Win32Resources** option are mutually exclusive. If you try to use both options in the same command line you will get a build error.

An application that has no application manifest that specifies a requested execution level will be subject to file and registry virtualization under the User Account Control feature in Windows. For more information, see User Account Control.

Your application will be subject to virtualization if either of these conditions is true:

- You use the **NoWin32Manifest** option and you don't provide a manifest in a later build step or as part of a Windows Resource (*.res*) file by using the **Win32Resource** option.
- You provide a custom manifest that doesn't specify a requested execution level.

Visual Studio creates a default *.manifest* file and stores it in the debug and release directories alongside the executable file. You can add a custom manifest by creating one in any text editor and then adding the file to the project. Or, you can right-click the **Project** icon in **Solution Explorer**, select **Add New Item**, and then select **Application Manifest File**. After you've added your new or existing manifest file, it will appear in the **Manifest** drop down list. For more information, see Application Page, Project Designer (C#).

You can provide the application manifest as a custom post-build step or as part of a Win32 resource file by using the **NoWin32Manifest** option. Use that same option if you want your application to be subject to file or registry virtualization on Windows Vista.

NoWin32Manifest

Use the **NoWin32Manifest** option to instruct the compiler not to embed any application manifest into the executable file.

XML

```
<NoWin32Manifest />
```

When this option is used, the application will be subject to virtualization on Windows Vista unless you provide an application manifest in a Win32 Resource file or during a later build step.

In Visual Studio, set this option in the **Application Property** page by selecting the **Create Application Without a Manifest** option in the **Manifest** drop down list. For more information, see Application Page, Project Designer (C#).

Resources

Embeds the specified resource into the output file.

XML

```
<Resources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</Resources>
```

`filename` is the .NET resource file that you want to embed in the output file. `identifier` (optional) is the logical name for the resource; the name that is used to load the

resource. The default is the name of the file. `accessibility-modifier` (optional) is the accessibility of the resource: public or private. The default is public. By default, resources are public in the assembly when they're created by using the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other accessibility other than `public` or `private` is allowed. If `filename` is a .NET resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the System.Resources namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource` methods in the Assembly class to access the resource at run time. The order of the resources in the output file is determined from the order specified in the project file.

LinkResources

Creates a link to a .NET resource in the output file. The resource file isn't added to the output file. **LinkResources** differs from the **Resource** option, which does embed a resource file in the output file.

XML

```
<LinkResources Include=filename>
  <LogicalName>identifier</LogicalName>
  <Access>accessibility-modifier</Access>
</LinkResources>
```

`filename` is the .NET resource file to which you want to link from the assembly. `identifier` (optional) is the logical name for the resource; the name that is used to load the resource. The default is the name of the file. `accessibility-modifier` (optional) is the accessibility of the resource: public or private. The default is public. By default, linked resources are public in the assembly when they're created with the C# compiler. To make the resources private, specify `private` as the accessibility modifier. No other modifier other than `public` or `private` is allowed. If `filename` is a .NET resource file created, for example, by Resgen.exe or in the development environment, it can be accessed with members in the System.Resources namespace. For more information, see System.Resources.ResourceManager. For all other resources, use the `GetManifestResource` methods in the Assembly class to access the resource at run time. The file specified in `filename` can be any format. For example, you may want to make a native DLL part of the assembly, so that it can be installed into the global assembly cache and accessed from managed code in the assembly. You can do the same thing in

the Assembly Linker. For more information, see Al.exe (Assembly Linker) and Working with Assemblies and the Global Assembly Cache.

Miscellaneous C# Compiler Options

Article • 01/18/2023

The following options control miscellaneous compiler behavior. The new MSBuild syntax is shown in **Bold**. The older *csc.exe* command-line syntax is shown in `code style`.

- **ResponseFiles** / `@CustomOpts.RSP` : Read the specified response file for more options.
- **NoLogo** / `-nologo` : Suppress compiler copyright message.
- **NoConfig** / `-noconfig` : Don't auto include *CSC.RSP* file.

> **Note**
>
> Refer to Compiler options for more information on configuring these options for your project.

ResponseFiles

The **ResponseFiles** option lets you specify a file that contains compiler options and source code files to compile.

```
XML

<ResponseFiles>response_file</ResponseFiles>
```

The `response_file` specifies the file that lists compiler options or source code files to compile. The compiler options and source code files will be processed by the compiler as if they had been specified on the command line. To specify more than one response file in a compilation, specify multiple response file options. In a response file, multiple compiler options and source code files can appear on one line. A single compiler option specification must appear on one line (can't span multiple lines). Response files can have comments that begin with the # symbol. Specifying compiler options from within a response file is just like issuing those commands on the command line. The compiler processes the command options as they're read. Command-line arguments can override previously listed options in response files. Conversely, options in a response file will override options listed previously on the command line or in other response files. C# provides the csc.rsp file, which is located in the same directory as the csc.exe file. For more information about the response file format, see NoConfig. This compiler option

cannot be set in the Visual Studio development environment, nor can it be changed programmatically. The following are a few lines from a sample response file:

```
Console

# build the first output file
-target:exe -out:MyExe.exe source1.cs source2.cs
```

NoLogo

The **NoLogo** option suppresses display of the sign-on banner when the compiler starts up and display of informational messages during compiling.

```
XML

<NoLogo>true</NoLogo>
```

NoConfig

The **NoConfig** option tells the compiler not to compile with the *csc.rsp* file.

```
XML

<NoConfig>true</NoConfig>
```

The *csc.rsp* file references all the assemblies shipped with .NET Framework. The actual references that the Visual Studio .NET development environment includes depend on the project type. You can modify the *csc.rsp* file and specify additional compiler options that should be included in every compilation. If you don't want the compiler to look for and use the settings in the *csc.rsp* file, specify **NoConfig**. This compiler option is unavailable in Visual Studio and cannot be changed programmatically.

Advanced C# compiler options

08/22/2025

The following options support advanced scenarios. The new MSBuild syntax is shown in **Bold**. The older `csc.exe` syntax is shown in `code style`.

- **StartupObject** / `-main`: Specify the type that contains the entry point.
- **PdbFile** / `-pdb`: Specify debug information file name.
- **PathMap** / `-pathmap`: Specify a mapping for source path names output by the compiler.
- **ApplicationConfiguration** / `-appconfig`: Specify an application configuration file containing assembly binding settings.
- **AdditionalLibPaths** / `-lib`: Specify additional directories to search in for references.
- **GenerateFullPaths** / `-fullpath`: Compiler generates fully qualified paths.
- **PreferredUILang** / `-preferreduilang`: Specify the preferred output language name.
- **BaseAddress** / `-baseaddress`: Specify the base address for the library to be built.
- **ChecksumAlgorithm** / `-checksumalgorithm` : Specify algorithm for calculating source file checksum stored in PDB.
- **CodePage** / `-codepage`: Specify the codepage to use when opening source files.
- **Utf8Output** / `-utf8output`: Output compiler messages in UTF-8 encoding.
- **FileAlignment** / `-filealign`: Specify the alignment used for output file sections.
- **ErrorEndLocation** / `-errorendlocation`: Output line and column of the end location of each error.
- **NoStandardLib** / `-nostdlib`: Don't reference standard library. (Prevents automatic reference to the .NET Base Class Library - *mscorlib.dll* in .NET Framework projects, *System.Private.CoreLib.dll* in modern .NET projects.)
- **SubsystemVersion** / `-subsystemversion`: Specify subsystem version of this assembly.
- **ModuleAssemblyName** / `-moduleassemblyname`: Name of the assembly that this module will be a part of.
- **ReportIVTs** / `-reportivts`: Produce additional information on System.Runtime.CompilerServices.InternalsVisibleToAttribute information.

You add any of these options in a `<PropertyGroup>` element in your `*.csproj` file:

XML

```
<PropertyGroup>
    <StartupObject>...</StartupObject>
    ...
</PropertyGroup>
```

StartupObject

This option specifies the class that contains the entry point to the program when your project includes more than one type with a `Main` method. When there's exactly one acceptable `Main` method in the project, this option is ignored.

```
XML
```

```
<StartupObject>MyNamespace.Program</StartupObject>
```

Where `Program` is the type that contains the `Main` method. When this option is required (multiple `Main` methods exist), the provided class name must be fully qualified; it must include the full namespace containing the class, followed by the class name. For example, when the `Main` method is located inside the `Program` class in the `MyApplication.Core` namespace, the compiler option has to be `-main:MyApplication.Core.Program`. If the class is in the global namespace, the fully qualified name is the same as the simple class name (for example, `Program` instead of `MyNamespace.Program`).

> **Note**
>
> This option can't be used for a project that includes top-level statements, even if that project contains one or more `Main` methods.

PdbFile

The **PdbFile** compiler option specifies the name and location of the debug symbols file. The `filename` value specifies the name and location of the debug symbols file.

```
XML
```

```
<PdbFile>filename</PdbFile>
```

When you specify DebugType, the compiler creates a *.pdb* file in the same directory where the compiler creates the output file (.exe or .dll). The *.pdb* file has the same base file name as the name of the output file. **PdbFile** allows you to specify a nondefault file name and location for the .pdb file. This compiler option can't be set in the Visual Studio development environment, nor can it be changed programmatically.

PathMap

> ⓘ Note
>
> Specifying **PathMap** prevents breakpoints from working in local debug builds. Only set **PathMap** for production or continuous integration builds.

The **PathMap** compiler option specifies how to map physical paths to source path names output by the compiler. This option maps each physical path on the machine where the compiler runs to a corresponding path that should be written in the output files. In the following example, `path1` is the full path to the source files in the current environment, and `sourcePath1` is the source path substituted for `path1` in any output files. To specify multiple mapped source paths, separate each with a comma.

```
XML

<PathMap>path1=sourcePath1,path2=sourcePath2</PathMap>
```

The compiler writes the source path into its output for the following reasons:

1. The source path is substituted for an argument when the CallerFilePathAttribute is applied to an optional parameter.
2. The source path is embedded in a PDB file.
3. The path of the PDB file is embedded into a PE (portable executable) file.

ApplicationConfiguration

The **ApplicationConfiguration** compiler option enables a C# application to specify the location of an assembly's application configuration (app.config) file to the common language runtime (CLR) at assembly binding time.

```
XML

<ApplicationConfiguration>file</ApplicationConfiguration>
```

Where `file` is the application configuration file that contains assembly binding settings. One use of **ApplicationConfiguration** is advanced scenarios in which an assembly has to reference both the .NET Framework version and the .NET Framework for Silverlight version of a particular reference assembly at the same time. For example, a XAML designer written in Windows Presentation Foundation (WPF) might have to reference both the WPF Desktop, for the designer's user interface, and the subset of WPF that is included with Silverlight. The same designer assembly has to access both assemblies. By default, the separate references cause a compiler error, because assembly binding sees the two assemblies as equivalent. The

ApplicationConfiguration compiler option enables you to specify the location of an app.config file that disables the default behavior by using a `<supportPortability>` tag, as shown in the following example.

XML

```xml
<supportPortability PKT="7cec85d7bea7798e" enable="false"/>
```

The compiler passes the location of the file to the CLR's assembly-binding logic.

> **Note**
>
> To use the app.config file that is already set in the project, add property tag `<UseAppConfigForCompiler>` to the *.csproj* file and set its value to `true`. To specify a different app.config file, add property tag `<AppConfigForCompiler>` and set its value to the location of the file.

The following example shows an app.config file that enables an application to have references to both the .NET Framework implementation and the .NET Framework for Silverlight implementation of any .NET Framework assembly that exists in both implementations. The **ApplicationConfiguration** compiler option specifies the location of this app.config file.

XML

```xml
<configuration>
  <runtime>
    <assemblyBinding>
      <supportPortability PKT="7cec85d7bea7798e" enable="false"/>
      <supportPortability PKT="31bf3856ad364e35" enable="false"/>
    </assemblyBinding>
  </runtime>
</configuration>
```

AdditionalLibPaths

The **AdditionalLibPaths** option specifies the location of assemblies referenced with the References option.

XML

```xml
<AdditionalLibPaths>dir1[,dir2]</AdditionalLibPaths>
```

Where `dir1` is a directory for the compiler to look in if a referenced assembly isn't found in the current working directory (the directory from which you're invoking the compiler) or in the common language runtime's system directory. `dir2` is one or more additional directories to search in for assembly references. Separate directory names with a comma, and without white space between them. The compiler searches for assembly references that aren't fully qualified in the following order:

1. Current working directory.
2. The common language runtime system directory.
3. Directories specified by **AdditionalLibPaths**.
4. Directories specified by the LIB environment variable.

Use **Reference** to specify an assembly reference. **AdditionalLibPaths** is additive. Specifying it more than once appends to any prior values. Since the path to the dependent assembly isn't specified in the assembly manifest, the application will find and use the assembly in the global assembly cache. The compiler referencing the assembly doesn't imply the common language runtime can find and load the assembly at run time. See How the Runtime Locates Assemblies for details on how the runtime searches for referenced assemblies.

GenerateFullPaths

The **GenerateFullPaths** option causes the compiler to specify the full path to the file when listing compilation errors and warnings.

Xml

```
<GenerateFullPaths>true</GenerateFullPaths>
```

By default, errors and warnings that result from compilation specify the name of the file in which an error was found. The **GenerateFullPaths** option causes the compiler to specify the full path to the file. This compiler option is unavailable in Visual Studio and can't be changed programmatically.

PreferredUILang

By using the **PreferredUILang** compiler option, you can specify the language in which the C# compiler displays output, such as error messages.

XML

```
<PreferredUILang>language</PreferredUILang>
```

Where `language` is the language name of the language to use for compiler output. You can use the **PreferredUILang** compiler option to specify the language that you want the C# compiler to use for error messages and other command-line output. If the language pack for the language isn't installed, the language setting of the operating system is used instead.

BaseAddress

The **BaseAddress** option lets you specify the preferred base address at which to load a DLL. For more information about when and why to use this option, see Larry Osterman's WebLog.

```
XML

<BaseAddress>address</BaseAddress>
```

Where `address` is the base address for the DLL. This address can be specified as a decimal, hexadecimal, or octal number. The default base address for a DLL is set by the .NET common language runtime. The lower-order word in this address will be rounded. For example, if you specify `0x11110001`, it is rounded to `0x11110000`. To complete the signing process for a DLL, use SN.EXE with the -R option.

ChecksumAlgorithm

This option controls the checksum algorithm we use to encode source files in the PDB.

```
XML

<ChecksumAlgorithm>algorithm</ChecksumAlgorithm>
```

The `algorithm` must be either `SHA1` (default) or `SHA256`.

CodePage

This option specifies which codepage to use during compilation if the required page isn't the current default codepage for the system.

```
XML

<CodePage>id</CodePage>
```

Where `id` is the ID of the code page to use for all source code files in the compilation. The compiler first attempts to interpret all source files as UTF-8. If your source code files are in an encoding other than UTF-8 and use characters other than 7-bit ASCII characters, use the **CodePage** option to specify which code page should be used. **CodePage** applies to all source code files in your compilation. See GetCPInfo for information on how to find which code pages are supported on your system.

Utf8Output

The **Utf8Output** option displays compiler output using UTF-8 encoding.

XML

```
<Utf8Output>true</Utf8Output>
```

In some international configurations, compiler output can't correctly be displayed in the console. Use **Utf8Output** and redirect compiler output to a file.

FileAlignment

The **FileAlignment** option lets you specify the size of sections in your output file. Valid values are 512, 1024, 2048, 4096, and 8192. These values are in bytes.

XML

```
<FileAlignment>number</FileAlignment>
```

You set the **FileAlignment** option from the **Advanced** page of the **Build** properties for your project in Visual Studio. Each section is aligned on a boundary that is a multiple of the **FileAlignment** value. There's no fixed default. If **FileAlignment** isn't specified, the common language runtime picks a default at compile time. By specifying the section size, you affect the size of the output file. Modifying section size may be useful for programs that run on smaller devices. Use DUMPBIN to see information about sections in your output file.

ErrorEndLocation

Instructs the compiler to output line and column of the end location of each error.

XML

```
<ErrorEndLocation>true</ErrorEndLocation>
```

By default, the compiler writes the starting location in source for all errors and warnings. When this option is set to true, the compiler writes both the starting and end location for each error and warning.

NoStandardLib

NoStandardLib prevents the automatic reference to the .NET Base Class Library. In .NET Framework projects, this refers to *mscorlib.dll*, while in modern .NET projects, this refers to *System.Private.CoreLib.dll*. Both assemblies define the entire System namespace.

XML

```
<NoStandardLib>true</NoStandardLib>
```

Use this option if you want to define or create your own System namespace and objects. If you don't specify **NoStandardLib**, the appropriate base class library is automatically referenced (same as specifying `<NoStandardLib>false</NoStandardLib>`).

SubsystemVersion

Specifies the minimum version of the subsystem on which the executable file runs. Most commonly, this option ensures that the executable file can use security features that aren't available with older versions of Windows.

> **Note**
>
> To specify the subsystem itself, use the **OutputType** compiler option.

XML

```
<SubsystemVersion>major.minor</SubsystemVersion>
```

The `major.minor` specify the minimum required version of the subsystem, as expressed in a dot notation for major and minor versions. For example, you can specify that an application can't run on an operating system that's older than Windows 7. Set the value of this option to 6.01, as the table later in this article describes. You specify the values for `major` and `minor` as integers. Leading zeroes in the `minor` version don't change the version, but trailing zeroes do. For

example, 6.1 and 6.01 refer to the same version, but 6.10 refers to a different version. We recommend expressing the minor version as two digits to avoid confusion.

The following table lists common subsystem versions of Windows.

Expand table

Windows version	Subsystem version
Windows Server 2003	5.02
Windows Vista	6.00
Windows 7	6.01
Windows Server 2008	6.01
Windows 8	6.02
Windows 8.1	6.03
Windows 10	6.04
Windows 11	6.04

The default value of the **SubsystemVersion** compiler option depends on the conditions in the following list:

- The default value is 6.02 if any compiler option in the following list is set:
  - -target:appcontainerexe
  - -target:winmdobj
  - -platform:arm
- The default value is 6.00 if you're using MSBuild, you're targeting .NET Framework 4.5 or later, and you haven't set any of the compiler options that were specified earlier in this list.
- For modern .NET projects, the default value is 6.00, which allows your application to run on Windows Vista and later versions.
- The default value is 4.00 if none of the previous conditions are true.

ModuleAssemblyName

Specifies the name of an assembly whose nonpublic types a *.netmodule* can access.

XML

```
<ModuleAssemblyName>assembly_name</ModuleAssemblyName>
```

ModuleAssemblyName should be used when building a *.netmodule*, and where the following conditions are true:

- The *.netmodule* needs access to nonpublic types in an existing assembly.
- You know the name of the assembly into which the .netmodule will be built.
- The existing assembly has granted friend assembly access to the assembly into which the *.netmodule* will be built.

For more information on building a .netmodule, see **OutputType** option of **module**. For more information on friend assemblies, see Friend Assemblies.

ReportIVTs

Enable or disable additional diagnostic information about System.Runtime.CompilerServices.InternalsVisibleToAttribute found during the compilation:

XML

```
<ReportIVTs>true</ReportIVTs>
```

The diagnostics are enabled if the element content is `true`, disabled if `false`, or not present.

ReportIVTs reports the following information when enabled:

1. Any inaccessible member diagnostics include their source assembly, if different than the current assembly.
2. The compiler prints the assembly identity of the project being compiled, its assembly name and public key.
3. For every reference passed to the compiler, it prints;
   a. The assembly identity of the reference
   b. Whether the reference grants the current project `InternalsVisibleTo`
   c. The name and all public keys of any assemblies granted `InternalsVisibleTo` from this assembly

Generate XML API documentation comments

Article • 02/19/2025

C# source files can include structured comments that produce API documentation for the types defined in those files. The C# compiler produces an *XML* file that contains structured data representing the comments and the API signatures. Other tools can process that XML output to create human-readable documentation in the form of web pages or PDF files, for example.

This process provides many advantages for you to add API documentation in your code:

- The C# compiler combines the structure of the C# code with the text of the comments into a single XML document.
- The C# compiler verifies that the comments match the API signatures for relevant tags.
- Tools that process the XML documentation files can define XML elements and attributes specific to those tools.

Tools like Visual Studio provide IntelliSense for many common XML elements used in documentation comments.

This article covers these topics:

- Documentation comments and XML file generation
- Tags validated by the C# compiler and Visual Studio
- Format of the generated XML file

Create XML documentation output

You create documentation for your code by writing special comment fields indicated by triple slashes. The comment fields include XML elements that describe the code block that follows the comments. For example:

C#

```
/// <summary>
/// This class performs an important function.
/// </summary>
public class MyClass { }
```

You set either the **GenerateDocumentationFile** or **DocumentationFile** option, and the compiler finds all comment fields with XML tags in the source code and creates an XML documentation file from those comments. When this option is enabled, the compiler generates the CS1591 warning for any publicly visible member declared in your project without XML documentation comments.

XML comment formats

The use of XML doc comments requires delimiters that indicate where a documentation comment begins and ends. You use the following delimiters with the XML documentation tags:

- `///` Single-line delimiter: The documentation examples and C# project templates use this form. If white space follows the delimiter, it isn't included in the XML output.

  > **Note**
  >
  > Visual Studio automatically inserts the `<summary>` and `</summary>` tags and positions your cursor within these tags after you type the `///` delimiter in the code editor. You can turn this feature on or off in the Options dialog box.

- `/** */` Multiline delimiters: The `/** */` delimiters have the following formatting rules:

  - On the line that contains the `/**` delimiter, if the rest of the line is white space, the line isn't processed for comments. If the first character after the `/**` delimiter is white space, that white-space character is ignored and the rest of the line is processed. Otherwise, the entire text of the line after the `/**` delimiter is processed as part of the comment.

  - On the line that contains the `*/` delimiter, if there's only white space up to the `*/` delimiter, that line is ignored. Otherwise, the text on the line up to the `*/` delimiter is processed as part of the comment.

  - For the lines after the one that begins with the `/**` delimiter, the compiler looks for a common pattern at the beginning of each line. The pattern can consist of optional white space and/or an asterisk (`*`), followed by more optional white space. If the compiler finds a common pattern at the beginning of each line that doesn't begin with the `/**` delimiter or end with the `*/` delimiter, it ignores that pattern for each line.

- The only part of the following comment processed is the line that begins with `<summary>`. The three tag formats produce the same comments.

```
C#

/** <summary>text</summary> */

/**
<summary>text</summary>
*/

/**
* <summary>text</summary>
*/
```

- The compiler identifies a common pattern of " * " at the beginning of the second and third lines. The pattern isn't included in the output.

```
C#

/**
* <summary>
* text </summary>*/
```

- The compiler finds no common pattern in the following comment because the second character on the third line isn't an asterisk. All text on the second and third lines is processed as part of the comment.

```
C#

/**
* <summary>
   text </summary>
*/
```

- The compiler finds no pattern in the following comment for two reasons. First, the number of spaces before the asterisk isn't consistent. Second, the fifth line begins with a tab, which doesn't match spaces. All text from lines two through five is processed as part of the comment.

```
C#

/**
  * <summary>
  * text
*  text2
```

```
     *  </summary>
 */
```

To refer to XML elements (for example, your function processes specific XML elements that you want to describe in an XML documentation comment), you can use the standard quoting mechanism (`<` and `>`). To refer to generic identifiers in code reference (`cref`) elements, you can use either the escape characters (for example, `cref="List<T>"`) or braces (`cref="List{T}"`). As a special case, the compiler parses the braces as angle brackets to make the documentation comment less cumbersome to the author when referring to generic identifiers.

> **Note**
>
> If you write comments using the single line XML comment delimiter, `///`, but don't include any tags, the compiler adds the text of those comments to the XML output file. However, the output doesn't include XML elements such as `<summary>`. Most tools that consume XML comments (including Visual Studio IntelliSense) don't read these comments.

Tools that accept XML documentation input

The following tools 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.

> **Note**
>
> The XML documentation comments aren't metadata; they aren't included in the compiled assembly and therefore they aren't accessible through reflection.

ID strings

Each type or member is stored in an element in the output XML file. Each of those elements has a unique ID string that identifies the type or member. The ID string must account for operators, parameters, return values, generic type parameters, `ref`, `in`, and `out` parameters. To encode all those potential elements, the compiler follows clearly defined rules for generating the ID strings. Programs that process the XML file use the ID string to identify the corresponding .NET metadata or reflection item that the documentation applies to.

The compiler observes the following rules when it generates the ID strings:

- No white space is in the string.
- The first part of the string identifies the kind of member using a single character followed by a colon. The following member types are used:

Expand table

Character	Member type	Notes
`N`	namespace	You can't add documentation comments to a namespace, but you can make `cref` references to them, where supported.
`T`	type	A type is a class, interface, struct, enum, or delegate.
`F`	field	
`P`	property	Includes indexers or other indexed properties.
`M`	method	Includes special methods, such as constructors and operators.
`E`	event	
`!`	error string	The rest of the string provides information about the error. The C# compiler generates error information for links that can't be

Character	Member type	Notes
		resolved.

- The second part of the string is the fully qualified name of the item, starting at the root of the namespace. The name of the item, its enclosing type(s), and namespace are separated by periods. If the name of the item itself has periods, they're replaced with the hash-sign ('#'). The grammar assumes that no item has a hash-sign directly in its name. For example, the fully qualified name of the String constructor is "System.String.#ctor".

- For properties and methods, the parameter list enclosed in parentheses follows. If there are no parameters, no parentheses are present. The parameters are separated by commas. The encoding of each parameter follows directly how it's encoded in a .NET signature (See Microsoft.VisualStudio.CorDebugInterop.CorElementType for definitions of the all caps elements in the following list):
  - Base types. Regular types (`ELEMENT_TYPE_CLASS` or `ELEMENT_TYPE_VALUETYPE`) are represented as the fully qualified name of the type.
  - Intrinsic types (for example, `ELEMENT_TYPE_I4`, `ELEMENT_TYPE_OBJECT`, `ELEMENT_TYPE_STRING`, `ELEMENT_TYPE_TYPEDBYREF`, and `ELEMENT_TYPE_VOID`) are represented as the fully qualified name of the corresponding full type. For example, `System.Int32` or `System.TypedReference`.
  - `ELEMENT_TYPE_PTR` is represented as a '*' following the modified type.
  - `ELEMENT_TYPE_BYREF` is represented as a '@' following the modified type.
  - `ELEMENT_TYPE_CMOD_OPT` is represented as a '!' and the fully qualified name of the modifier class, following the modified type.
  - `ELEMENT_TYPE_SZARRAY` is represented as "[]" following the element type of the array.
  - `ELEMENT_TYPE_ARRAY` is represented as [*lower bound*:`size`,*lower bound*:`size`] where the number of commas is the rank - 1, and the lower bounds and size of each dimension, if known, are represented in decimal. The lower bound and size are omitted if they aren't specified. If the lower bound and size for a particular dimension is omitted, the ':' is omitted as well. For example, a two-dimensional array with 1 as the lower bounds and unspecified sizes is [1:,1:].

- For conversion operators only (`op_Implicit` and `op_Explicit`), the return value of the method is encoded as a `~` followed by the return type. For example: `<member name="M:System.Decimal.op_Explicit(System.Decimal arg)~System.Int32">` is the

tag for the cast operator `public static explicit operator int (decimal value);` declared in the `System.Decimal` class.

- For generic types, the name of the type is followed by a backtick and then a number that indicates the number of generic type parameters. For example: `<member name="T:SampleClass`2">` is the tag for a type that is defined as `public class SampleClass<T, U>`. For methods that take generic types as parameters, the generic type parameters are specified as numbers prefaced with backticks (for example `0,`1). Each number represents a zero-based array notation for the type's generic parameters.
  - `ELEMENT_TYPE_PINNED` is represented as a '^' following the modified type. The C# compiler never generates this encoding.
  - `ELEMENT_TYPE_CMOD_REQ` is represented as a '|' and the fully qualified name of the modifier class, following the modified type. The C# compiler never generates this encoding.
  - `ELEMENT_TYPE_GENERICARRAY` is represented as "[?]" following the element type of the array. The C# compiler never generates this encoding.
  - `ELEMENT_TYPE_FNPTR` is represented as "=FUNC:`type`(*signature*)", where `type` is the return type, and *signature* is the arguments of the method. If there are no arguments, the parentheses are omitted. The C# compiler never generates this encoding.
  - The following signature components aren't represented because they aren't used to differentiate overloaded methods:
    - calling convention
    - return type
    - `ELEMENT_TYPE_SENTINEL`

The following examples show how the ID strings for a class and its members are generated:

C#

```
namespace MyNamespace;

/// <summary>
/// Enter description here for class X.
/// ID string generated is "T:MyNamespace.MyClass".
/// </summary>
public unsafe class MyClass
{
    /// <summary>
    /// Enter description here for the first constructor.
    /// ID string generated is "M:MyNamespace.MyClass.#ctor".
    /// </summary>
```

```
    public MyClass() { }

    /// <summary>
    /// Enter description here for the second constructor.
    /// ID string generated is "M:MyNamespace.MyClass.#ctor(System.Int32)".
    /// </summary>
    /// <param name="i">Describe parameter.</param>
    public MyClass(int i) { }

    /// <summary>
    /// Enter description here for field Message.
    /// ID string generated is "F:MyNamespace.MyClass.Message".
    /// </summary>
    public string? Message;

    /// <summary>
    /// Enter description for constant PI.
    /// ID string generated is "F:MyNamespace.MyClass.PI".
    /// </summary>
    public const double PI = 3.14;

    /// <summary>
    /// Enter description for method Func.
    /// ID string generated is "M:MyNamespace.MyClass.Func".
    /// </summary>
    /// <returns>Describe return value.</returns>
    public int Func() => 1;

    /// <summary>
    /// Enter description for method SomeMethod.
    /// ID string generated is
"M:MyNamespace.MyClass.SomeMethod(System.String,System.Int32@,System.Void*)"
.
    /// </summary>
    /// <param name="str">Describe parameter.</param>
    /// <param name="num">Describe parameter.</param>
    /// <param name="ptr">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public int SomeMethod(string str, ref int num, void* ptr) { return 1; }

    /// <summary>
    /// Enter description for method AnotherMethod.
    /// ID string generated is
"M:MyNamespace.MyClass.AnotherMethod(System.Int16[],System.Int32[0:,0:])".
    /// </summary>
    /// <param name="array1">Describe parameter.</param>
    /// <param name="array">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public int AnotherMethod(short[] array1, int[,] array) { return 0; }

    /// <summary>
    /// Enter description for operator.
    /// ID string generated is
"M:MyNamespace.MyClass.op_Addition(MyNamespace.MyClass,MyNamespace.MyClass)"
.
```

```
    /// </summary>
    /// <param name="first">Describe parameter.</param>
    /// <param name="second">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public static MyClass operator +(MyClass first, MyClass second) { return
first; }

    /// <summary>
    /// Enter description for property.
    /// ID string generated is "P:MyNamespace.MyClass.Prop".
    /// </summary>
    public int Prop { get { return 1; } set { } }

    /// <summary>
    /// Enter description for event.
    /// ID string generated is "E:MyNamespace.MyClass.OnHappened".
    /// </summary>
    public event Del? OnHappened;

    /// <summary>
    /// Enter description for index.
    /// ID string generated is "P:MyNamespace.MyClass.Item(System.String)".
    /// </summary>
    /// <param name="str">Describe parameter.</param>
    /// <returns></returns>
    public int this[string s] => 1;

    /// <summary>
    /// Enter description for class Nested.
    /// ID string generated is "T:MyNamespace.MyClass.Nested".
    /// </summary>
    public class Nested { }

    /// <summary>
    /// Enter description for delegate.
    /// ID string generated is "T:MyNamespace.MyClass.Del".
    /// </summary>
    /// <param name="i">Describe parameter.</param>
    public delegate void Del(int i);

    /// <summary>
    /// Enter description for operator.
    /// ID string generated is
"M:MyNamespace.MyClass.op_Explicit(MyNamespace.MyClass)~System.Int32".
    /// </summary>
    /// <param name="myParameter">Describe parameter.</param>
    /// <returns>Describe return value.</returns>
    public static explicit operator int(MyClass myParameter) => 1;
}
```

C# language specification

For more information, see the C# Language Specification annex on documentation comments.

Recommended XML tags for C# documentation comments

07/03/2025

C# documentation comments use XML elements to define the structure of the output documentation. One consequence of this feature is that you can add any valid XML in your documentation comments. The C# compiler copies these elements into the output XML file. While you can use any valid XML in your comments (including any valid HTML element), documenting code is recommended for many reasons.

What follows are some recommendations, general use case scenarios, and things that you should know when using XML documentation tags in your C# code. While you can put any tags into your documentation comments, this article describes the recommended tags for the most common language constructs. In all cases, you should adhere to these recommendations:

- For the sake of consistency, all publicly visible types and their public members should be documented.
- Private members can also be documented using XML comments. However, it exposes the inner (potentially confidential) workings of your library.
- At a bare minimum, types and their members should have a `<summary>` tag.
- Documentation text should be written using complete sentences ending with full stops.
- Partial classes are fully supported, and documentation information is concatenated into a single entry for each type. If both declarations of a partial member have documentation comments, the comments on the implementing declaration are written to the output XML.

XML documentation starts with `///`. When you create a new project, the templates put some starter `///` lines in for you. The processing of these comments has some restrictions:

- The documentation must be well-formed XML. If the XML isn't well formed, the compiler generates a warning. The documentation file contains a comment that says that an error was encountered.
- Some of the recommended tags have special meanings:
  - The `<param>` tag is used to describe parameters. If used, the compiler verifies that the parameter exists and that all parameters are described in the documentation. If the verification fails, the compiler issues a warning.
  - The `cref` attribute can be attached to any tag to reference a code element. The compiler verifies that this code element exists. If the verification fails, the compiler issues a warning. The compiler respects any `using` directives when it looks for a type described in the `cref` attribute.

    - The `<summary>` tag is used by IntelliSense inside Visual Studio to display additional information about a type or member.

      > **Note**
      >
      > The XML file does not provide full information about the type and members (for example, it does not contain any type information). To get full information about a type or member, use the documentation file together with reflection on the actual type or member.

- Developers are free to create their own set of tags. The compiler copies these tags to the output file.

Some of the recommended tags can be used on any language element. Others have more specialized usage. Finally, some of the tags are used to format text in your documentation. This article describes the recommended tags organized by their use.

The compiler verifies the syntax of the elements followed by a single * in the following list. Visual Studio provides IntelliSense for the tags verified by the compiler and all tags followed by ** in the following list. In addition to the tags listed here, the compiler and Visual Studio validate the `<b>`, `<i>`, `<u>`, `<br/>`, and `<a>` tags. The compiler also validates `<tt>`, which is deprecated HTML.

> **Note**
>
> HTML tags like `<br/>` are useful for formatting within documentation comments. The `<br/>` tag creates line breaks, while other HTML tags provide text formatting. These tags work in IntelliSense tooltips and generated documentation.

- General Tags used for multiple elements - These tags are the minimum set for any API.
    - <summary>: The value of this element is displayed in IntelliSense in Visual Studio.
    - <remarks> **
- Tags used for members - These tags are used when documenting methods and properties.
    - <returns>: The value of this element is displayed in IntelliSense in Visual Studio.
    - <param> *: The value of this element is displayed in IntelliSense in Visual Studio.
    - <paramref>
    - <exception> *
    - <value>: The value of this element is displayed in IntelliSense in Visual Studio.
- Format documentation output - These tags provide formatting directions for tools that generate documentation.

    - <para>
    - <list>
    - <c>
    - <code>
    - <example> **
    - <b>
    - <i>
    - <u>
    -
    - <a>
- Reuse documentation text - These tags provide tools that make it easier to reuse XML comments.
    - <inheritdoc> **
    - <include> *
- Generate links and references - These tags generate links to other documentation.
    - <see> *
    - <seealso> *
    - cref
    - href
- Tags for generic types and methods - These tags are used only on generic types and methods
    - <typeparam> *: The value of this element is displayed in IntelliSense in Visual Studio.
    - <typeparamref>

> ⓘ **Note**
>
> Documentation comments cannot be applied to a namespace.

If you want angle brackets to appear in the text of a documentation comment, use the HTML encoding of `<` and `>`, which is `<` and `>` respectively. This encoding is shown in the following example.

C#

```
/// <summary>
/// This property always returns a value &lt; 1.
/// </summary>
```

General tags

<summary>

```xml
<summary>description</summary>
```

The `<summary>` tag should be used to describe a type or a type member. Use <remarks> to add supplemental information to a type description. Use the cref attribute to enable documentation tools such as DocFX and Sandcastle to create internal hyperlinks to documentation pages for code elements. The text for the `<summary>` tag is displayed in IntelliSense and in the Object Browser window.

<remarks>

```xml
<remarks>
description
</remarks>
```

The `<remarks>` tag is used to add information about a type or a type member, supplementing the information specified with <summary>. This information is displayed in the Object Browser window. This tag can include more lengthy explanations. You might find that using `CDATA` sections for markdown make writing it more convenient. Tools such as docfx process the markdown text in `CDATA` sections.

Document members

<returns>

```xml
<returns>description</returns>
```

The `<returns>` tag should be used in the comment for a method declaration to describe the return value.

<param>

```xml
```

```
<param name="name">description</param>
```

- `name`: The name of a method parameter. Enclose the name in quotation marks ("). The names for parameters must match the API signature. If one or more parameters aren't covered, the compiler issues a warning. The compiler also issues a warning if the value of `name` doesn't match a formal parameter in the method declaration.

The `<param>` tag should be used in the comment for a method declaration to describe one of the parameters for the method. To document multiple parameters, use multiple `<param>` tags. The text for the `<param>` tag is displayed in IntelliSense, the Object Browser, and the Code Comment Web Report.

<paramref>

XML

```
<paramref name="name"/>
```

- `name`: The name of the parameter to refer to. Enclose the name in quotation marks (").

The `<paramref>` tag gives you a way to indicate that a word in the code comments, for example in a `<summary>` or `<remarks>` block refers to a parameter. The XML file can be processed to format this word in some distinct way, such as with a bold or italic font.

<exception>

XML

```
<exception cref="member">description</exception>
```

- cref = "`member`": A reference to an exception that is available from the current compilation environment. The compiler checks that the given exception exists and translates `member` to the canonical element name in the output XML. `member` must appear within quotation marks (").

The `<exception>` tag lets you specify which exceptions can be thrown. This tag can be applied to definitions for methods, properties, events, and indexers.

<value>

XML

```
<value>property-description</value>
```

The `<value>` tag lets you describe the value that a property represents. When you add a property via code wizard in the Visual Studio .NET development environment, it adds a <summary> tag for the new property. You manually add a `<value>` tag to describe the value that the property represents.

Format documentation output

<para>

XML

```
<remarks>
    <para>
        This is an introductory paragraph.
    </para>
    <para>
        This paragraph contains more details.
    </para>
</remarks>
```

The `<para>` tag is for use inside a tag, such as <summary>, <remarks>, or <returns>, and lets you add structure to the text. The `<para>` tag creates a double spaced paragraph. Use the `<br/>` tag if you want a single spaced paragraph.

Here's an example showing the difference between `<para>` and `<br/>`:

C#

```
/// <summary>
/// Example using para tags:
/// <para>This is the first paragraph.</para>
/// <para>This is the second paragraph with double spacing.</para>
///
/// Example using br tags:
/// First line of text<br/>
/// Second line of text with single spacing<br/>
/// Third line of text
/// </summary>
public void FormattingExample()
{
```

```
    // This method demonstrates paragraph and line break formatting
}
```

<list>

XML

```
<list type="bullet|number|table">
    <listheader>
        <term>term</term>
        <description>description</description>
    </listheader>
    <item>
        <term>Assembly</term>
        <description>The library or executable built from a compilation.
</description>
    </item>
</list>
```

The `<listheader>` block is used to define the heading row of either a table or definition list.

When defining a table:

- You only need to supply an entry for `term` in the heading.
- Each item in the list is specified with an `<item>` block. For each `item`, you only need to supply an entry for `description`.

When creating a definition list:

- You must supply an entry for `term` in the heading.
- Each item in the list is specified with an `<item>` block. Each `item` must contain both a `term` and `description`.

A list or table can have as many `<item>` blocks as needed.

<c>

XML

```
<c>text</c>
```

The `<c>` tag gives you a way to indicate that text within a description should be marked as code. Use <code> to indicate multiple lines as code.

<code>

XML

```
<code>
    var index = 5;
    index++;
</code>
```

The `<code>` tag is used to indicate multiple lines of code. Use `<c>` to indicate that single-line text within a description should be marked as code.

<example>

XML

```
<example>
This shows how to increment an integer.
<code>
    var index = 5;
    index++;
</code>
</example>
```

The `<example>` tag lets you specify an example of how to use a method or other library member. An example commonly involves using the `<code>` tag.

<b>

XML

```
<b>text</b>
```

The `<b>` tag is used to make text bold within documentation comments. This HTML formatting tag is validated by the compiler and Visual Studio, and the formatted text appears in IntelliSense and generated documentation.

<i>

XML

```
<i>text</i>
```

The `<i>` tag is used to make text italic within documentation comments. This HTML formatting tag is validated by the compiler and Visual Studio, and the formatted text appears in IntelliSense and generated documentation.

<u>

XML

```
<u>text</u>
```

The `<u>` tag is used to underline text within documentation comments. This HTML formatting tag is validated by the compiler and Visual Studio, and the formatted text appears in IntelliSense and generated documentation.

XML

```
Line one<br/>Line two
```

The `<br/>` tag is used to insert a line break within documentation comments. Use this tag when you want a single spaced paragraph, as opposed to the `<para>` tag which creates double spaced paragraphs.

<a>

XML

```
<a href="https://example.com">Link text</a>
```

The `<a>` tag is used to create hyperlinks within documentation comments. The `href` attribute specifies the URL to link to. This HTML formatting tag is validated by the compiler and Visual Studio.

> **Note**
>
> The compiler also validates the `<tt>` tag, which is deprecated HTML. Use the <c> tag instead for inline code formatting.

Reuse documentation text

<inheritdoc>

```XML
<inheritdoc [cref=""] [path=""]/>
```

Inherit XML comments from base classes, interfaces, and similar methods. Using `inheritdoc` eliminates unwanted copying and pasting of duplicate XML comments and automatically keeps XML comments synchronized. When you add the `<inheritdoc>` tag to a type, all members inherit the comments as well.

- `cref`: Specify the member to inherit documentation from. Already defined tags on the current member aren't overridden by the inherited ones.
- `path`: The XPath expression query that results in a node set to show. You can use this attribute to filter the tags to include or exclude from the inherited documentation.

Add your XML comments in base classes or interfaces and let inheritdoc copy the comments to implementing classes. Add your XML comments to your synchronous methods and let inheritdoc copy the comments to your asynchronous versions of the same methods. If you want to copy the comments from a specific member, you use the `cref` attribute to specify the member.

<include>

```XML
<include file='filename' path='tagpath[@name="id"]' />
```

- `filename`: The name of the XML file containing the documentation. The file name can be qualified with a path relative to the source code file. Enclose `filename` in single quotation marks (' ').
- `tagpath`: The path of the tags in `filename` that leads to the tag `name`. Enclose the path in single quotation marks (' ').
- `name`: The name specifier in the tag that precedes the comments; `name` has an `id`.
- `id`: The ID for the tag that precedes the comments. Enclose the ID in quotation marks (").

The `<include>` tag lets you refer to comments in another file that describe the types and members in your source code. Including an external file is an alternative to placing

documentation comments directly in your source code file. By putting the documentation in a separate file, you can apply source control to the documentation separately from the source code. One person can have the source code file checked out and someone else can have the documentation file checked out. The `<include>` tag uses the XML XPath syntax. Refer to XPath documentation for ways to customize your `<include>` use.

For example, the following source code uses the `<include>` tag to include remarks. The file path is relative to the source.

C#

```
namespace MyNamespace;

public class MyType
{
    /// <returns>This is the returns text of MyMethod. It comes from triple slash
comments.</returns>
    /// <remarks>This is the remarks text of MyMethod. It comes from triple slash
comments.</remarks>
    /// <include file="MyAssembly.xml"
path="doc/members/member[@name='M:MyNamespace.MyType.MyMethod']/*" />
    public int MyMethod(int p) => p;
}
```

The XML source for the include file is shown in the following sample. It's structured the same as the XML file generated by the C# compiler. The XML file can contain text for multiple methods or types, as long as an XPath expression can identify them.

XML

```
<?xml version="1.0"?>
<doc>
    <members>
        <member name="M:MyNamespace.MyType.MyMethod">
            <param name="p">This is the description of the parameter p of
MyMethod. It comes from the included file.</param>
            <summary>This is the summary of MyMethod. It comes from the included
file.</summary>
        </member>
    </members>
</doc>
```

The XML output for this method is shown in the following example:

XML

```
<member name="M:MyNamespace.MyType.MyMethod(System.Int32)">
    <summary>This is the summary of MyMethod. It comes from the included file.
```

```
</summary>
    <returns>This is the returns text of MyMethod. It comes from triple slash
comments.</returns>
    <remarks>This is the remarks text of MyMethod. It comes from triple slash
comments.</remarks>
    <param name="p">This is the description of the parameter p of MyMethod. It
comes from the included file.</param>
</member>
```

> **Tip**
>
> The .NET Runtime team uses the `<include>` tag extensively in its documentation. You can see many examples by searching the dotnet/runtime repository.

Generate links and references

<see>

XML

```
<see cref="member"/>
<!-- or -->
<see cref="member">Link text</see>
<!-- or -->
<see href="link">Link Text</see>
<!-- or -->
<see langword="keyword"/>
```

- `cref="member"`: A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. Place *member* within quotation marks ("). You can provide different link text for a "cref", by using a separate closing tag.
- `href="link"`: A clickable link to a given URL. For example, `<see href="https://github.com">GitHub</see>` produces a clickable link with text GitHub that links to `https://github.com`. Use `href` instead of `cref` when linking to external web pages, as `cref` is designed for code references and won't create clickable links for external URLs.
- `langword="keyword"`: A language keyword, such as `true` or one of the other valid keywords.

The `<see>` tag lets you specify a link from within text. Use `<seealso>` to indicate that text should be placed in a See Also section. Use the cref attribute to create internal hyperlinks to documentation pages for code elements. You include the type parameters to specify a reference to a generic type or method, such as `cref="IDictionary{T, U}"`. Also, `href` is a valid attribute that functions as a hyperlink.

Here's an example showing the difference between `cref` and `href` when referencing external URLs:

C#

```
/// <summary>
/// This method demonstrates URL linking:
/// <see cref="https://learn.microsoft.com/dotnet/csharp"/> (won't create
clickable link)
/// <see href="https://learn.microsoft.com/dotnet/csharp">C# documentation</see>
(creates clickable link)
/// </summary>
public void UrlLinkingExample()
{
    // This method demonstrates the difference between cref and href for URLs
}
```

<seealso>

XML

```
<seealso cref="member"/>
<!-- or -->
<seealso href="link">Link Text</seealso>
```

- `cref="member"`: A reference to a member or field that is available to be called from the current compilation environment. The compiler checks that the given code element exists and passes `member` to the element name in the output XML. `member` must appear within quotation marks (").
- `href="link"`: A clickable link to a given URL. For example, `<seealso href="https://github.com">GitHub</seealso>` produces a clickable link with text GitHub that links to `https://github.com`.

The `<seealso>` tag lets you specify the text that you might want to appear in a **See Also** section. Use `<see>` to specify a link from within text. You can't nest the `seealso` tag inside the `summary` tag.

cref attribute

The `cref` attribute in an XML documentation tag means "code reference." It specifies that the inner text of the tag is a code element, such as a type, method, or property. Documentation tools like DocFX and Sandcastle use the `cref` attributes to automatically generate hyperlinks to the page where the type or member is documented.

href attribute

The `href` attribute means a reference to a web page. You can use it to directly reference online documentation about your API or library. When you need to link to external URLs in your documentation comments, use `href` instead of `cref` to ensure the links are clickable in IntelliSense tooltips and generated documentation.

Generic types and methods

<typeparam>

XML

```xml
<typeparam name="TResult">The type returned from this method</typeparam>
```

- `TResult`: The name of the type parameter. Enclose the name in quotation marks (").

The `<typeparam>` tag should be used in the comment for a generic type or method declaration to describe a type parameter. Add a tag for each type parameter of the generic type or method. The text for the `<typeparam>` tag is displayed in IntelliSense.

<typeparamref>

XML

```xml
<typeparamref name="TKey"/>
```

- `TKey`: The name of the type parameter. Enclose the name in quotation marks (").

Use this tag to enable consumers of the documentation file to format the word in some distinct way, for example in italics.

User-defined tags

All the tags outlined in this article represent those tags recognized by the C# compiler. However, a user is free to define their own tags. Tools like Sandcastle bring support for extra tags like <event> and <note>, and even support documenting namespaces. Custom or in-house documentation generation tools can also be used with the standard tags, and multiple output formats from HTML to PDF can be supported.

Example XML documentation comments

Article • 09/15/2021

This article contains three examples for adding XML documentation comments to most C# language elements. The first example shows how you document a class with different members. The second shows how you would reuse explanations for a hierarchy of classes or interfaces. The third shows tags to use for generic classes and members. The second and third examples use concepts that are covered in the first example.

Document a class, struct, or interface

The following example shows common language elements, and the tags you'll likely use to describe these elements. The documentation comments describe the use of the tags, rather than the class itself.

C#

```
/// <summary>
/// Every class and member should have a one sentence
/// summary describing its purpose.
/// </summary>
/// <remarks>
/// You can expand on that one sentence summary to
/// provide more information for readers. In this case,
/// the <c>ExampleClass</c> provides different C#
/// elements to show how you would add documentation
///comments for most elements in a typical class.
/// <para>
/// The remarks can add multiple paragraphs, so you can
/// write detailed information for developers that use
/// your work. You should add everything needed for
/// readers to be successful. This class contains
/// examples for the following:
/// </para>
/// <list type="table">
/// <item>
/// <term>Summary</term>
/// <description>
/// This should provide a one sentence summary of the class or member.
/// </description>
/// </item>
/// <item>
/// <term>Remarks</term>
/// <description>
/// This is typically a more detailed description of the class or member
/// </description>
/// </item>
/// <item>
```

```
/// <term>para</term>
/// <description>
/// The para tag separates a section into multiple paragraphs
/// </description>
/// </item>
/// <item>
/// <term>list</term>
/// <description>
/// Provides a list of terms or elements
/// </description>
/// </item>
/// <item>
/// <term>returns, param</term>
/// <description>
/// Used to describe parameters and return values
/// </description>
/// </item>
/// <item>
/// <term>value</term>
/// <description>Used to describe properties</description>
/// </item>
/// <item>
/// <term>exception</term>
/// <description>
/// Used to describe exceptions that may be thrown
/// </description>
/// </item>
/// <item>
/// <term>c, cref, see, seealso</term>
/// <description>
/// These provide code style and links to other
/// documentation elements
/// </description>
/// </item>
/// <item>
/// <term>example, code</term>
/// <description>
/// These are used for code examples
/// </description>
/// </item>
/// </list>
/// <para>
/// The list above uses the "table" style. You could
/// also use the "bullet" or "number" style. Neither
/// would typically use the "term" element.
/// <br/>
/// Note: paragraphs are double spaced. Use the *br*
/// tag for single spaced lines.
/// </para>
/// </remarks>
public class ExampleClass
{
    /// <value>
    /// The <c>Label</c> property represents a label
    /// for this instance.
```

```
    /// </value>
    /// <remarks>
    /// The <see cref="Label"/> is a <see langword="string"/>
    /// that you use for a label.
    /// <para>
    /// Note that there isn't a way to provide a "cref" to
    /// each accessor, only to the property itself.
    /// </para>
    /// </remarks>
    public string? Label
    {
        get;
        set;
    }

    /// <summary>
    /// Adds two integers and returns the result.
    /// </summary>
    /// <returns>
    /// The sum of two integers.
    /// </returns>
    /// <param name="left">
    /// The left operand of the addition.
    /// </param>
    /// <param name="right">
    /// The right operand of the addition.
    /// </param>
    /// <example>
    /// <code>
    /// int c = Math.Add(4, 5);
    /// if (c > 10)
    /// {
    ///     Console.WriteLine(c);
    /// }
    /// </code>
    /// </example>
    /// <exception cref="System.OverflowException">
    /// Thrown when one parameter is
    /// <see cref="Int32.MaxValue">MaxValue</see> and the other is
    /// greater than 0.
    /// Note that here you can also use
    /// <see
href="https://learn.microsoft.com/dotnet/api/system.int32.maxvalue"/>
    ///  to point a web page instead.
    /// </exception>
    /// <see cref="ExampleClass"/> for a list of all
    /// the tags in these examples.
    /// <seealso cref="ExampleClass.Label"/>
    public static int Add(int left, int right)
    {
        if ((left == int.MaxValue && right > 0) || (right == int.MaxValue &&
left > 0))
            throw new System.OverflowException();

        return left + right;
```

```
    }
}

/// <summary>
/// This is an example of a positional record.
/// </summary>
/// <remarks>
/// There isn't a way to add XML comments for properties
/// created for positional records, yet. The language
/// design team is still considering what tags should
/// be supported, and where. Currently, you can use
/// the "param" tag to describe the parameters to the
/// primary constructor.
/// </remarks>
/// <param name="FirstName">
/// This tag will apply to the primary constructor parameter.
/// </param>
/// <param name="LastName">
/// This tag will apply to the primary constructor parameter.
/// </param>
public record Person(string FirstName, string LastName);
```

Adding documentation can clutter your source code with large sets of comments intended for users of your library. You use the `<Include>` tag to separate your XML comments from your source. Your source code references an XML file with the `<Include>` tag:

C#

```
/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
    static void Main()
    {
    }
}

/// <include file='xml_include_tag.xml'
path='MyDocs/MyMembers[@name="test2"]/*' />
class Test2
{
    public void Test()
    {
    }
}
```

The second file, *xml_include_tag.xml*, contains the documentation comments.

XML

```
<MyDocs>
    <MyMembers name="test">
        <summary>
        The summary for this type.
        </summary>
    </MyMembers>
    <MyMembers name="test2">
        <summary>
        The summary for this other type.
        </summary>
    </MyMembers>
</MyDocs>
```

Document a hierarchy of classes and interfaces

The `<inheritdoc>` element means a type or member *inherits* documentation comments from a base class or interface. You can also use the `<inheritdoc>` element with the `cref` attribute to inherit comments from a member of the same type. The following example shows ways to use this tag. Note that when you add the `inheritdoc` attribute to a type, member comments are inherited. You can prevent the use of inherited comments by writing comments on the members in the derived type. Those will be chosen over the inherited comments.

C#

```
/// <summary>
/// A summary about this class.
/// </summary>
/// <remarks>
/// These remarks would explain more about this class.
/// In this example, these comments also explain the
/// general information about the derived class.
/// </remarks>
public class MainClass
{
}

///<inheritdoc/>
public class DerivedClass : MainClass
{
}

/// <summary>
/// This interface would describe all the methods in
/// its contract.
/// </summary>
/// <remarks>
/// While elided for brevity, each method or property
```

```
/// in this interface would contain docs that you want
/// to duplicate in each implementing class.
/// </remarks>
public interface ITestInterface
{
    /// <summary>
    /// This method is part of the test interface.
    /// </summary>
    /// <remarks>
    /// This content would be inherited by classes
    /// that implement this interface when the
    /// implementing class uses "inheritdoc"
    /// </remarks>
    /// <returns>The value of <paramref name="arg" /> </returns>
    /// <param name="arg">The argument to the method</param>
    int Method(int arg);
}

///<inheritdoc cref="ITestInterface"/>
public class ImplementingClass : ITestInterface
{
    // doc comments are inherited here.
    public int Method(int arg) => arg;
}

/// <summary>
/// This class shows hows you can "inherit" the doc
/// comments from one method in another method.
/// </summary>
/// <remarks>
/// You can inherit all comments, or only a specific tag,
/// represented by an xpath expression.
/// </remarks>
public class InheritOnlyReturns
{
    /// <summary>
    /// In this example, this summary is only visible for this method.
    /// </summary>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="/returns"/>
    public static bool MyChildMethod() { return false; }
}

/// <Summary>
/// This class shows an example ofsharing comments across methods.
/// </Summary>
public class InheritAllButRemarks
{
    /// <summary>
    /// In this example, this summary is visible on all the methods.
    /// </summary>
    /// <remarks>
    /// The remarks can be inherited by other methods
```

```
    /// using the xpath expression.
    /// </remarks>
    /// <returns>A boolean</returns>
    public static bool MyParentMethod(bool x) { return x; }

    /// <inheritdoc cref="MyParentMethod" path="//*[not(self::remarks)]"/>
    public static bool MyChildMethod() { return false; }
}
```

Generic types

Use the `<typeparam>` tag to describe type parameters on generic types and methods. The value for the `cref` attribute requires new syntax to reference a generic method or class:

C#

```
/// <summary>
/// This is a generic class.
/// </summary>
/// <remarks>
/// This example shows how to specify the <see cref="GenericClass{T}"/>
/// type as a cref attribute.
/// In generic classes and methods, you'll often want to reference the
/// generic type, or the type parameter.
/// </remarks>
class GenericClass<T>
{
    // Fields and members.
}

/// <Summary>
/// This shows examples of typeparamref and typeparam tags
/// </Summary>
public class ParamsAndParamRefs
{
    /// <summary>
    /// The GetGenericValue method.
    /// </summary>
    /// <remarks>
    /// This sample shows how to specify the <see cref="GetGenericValue"/>
    /// method as a cref attribute.
    /// The parameter and return value are both of an arbitrary type,
    /// <typeparamref name="T"/>
    /// </remarks>
    public static T GetGenericValue<T>(T para)
    {
        return para;
    }
}
```

Math class example

The following code shows a realistic example of adding doc comments to a math library.

C#

```
namespace TaggedLibrary
{
    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <summary>
    /// The main <c>Math</c> class.
    /// Contains all methods for performing basic math functions.
    /// <list type="bullet">
    /// <item>
    /// <term>Add</term>
    /// <description>Addition Operation</description>
    /// </item>
    /// <item>
    /// <term>Subtract</term>
    /// <description>Subtraction Operation</description>
    /// </item>
    /// <item>
    /// <term>Multiply</term>
    /// <description>Multiplication Operation</description>
    /// </item>
    /// <item>
    /// <term>Divide</term>
    /// <description>Division Operation</description>
    /// </item>
    /// </list>
    /// </summary>
    /// <remarks>
    /// <para>
    /// This class can add, subtract, multiply and divide.
    /// </para>
    /// <para>
    /// These operations can be performed on both
    /// integers and doubles.
    /// </para>
    /// </remarks>
    public class Math
    {
        // Adds two integers and returns the result
        /// <summary>
        /// Adds two integers <paramref name="a"/> and <paramref name="b"/>
        ///  and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two integers.
        /// </returns>
```

```
        /// <example>
        /// <code>
        /// int c = Math.Add(4, 5);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is <see cref="Int32.MaxValue"/> and
the other
        /// is greater than 0.
        /// </exception>
        /// See <see cref="Math.Add(double, double)"/> to add doubles.
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) ||
                (b == int.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }
            return a + b;
        }

        // Adds two doubles and returns the result
        /// <summary>
        /// Adds two doubles <paramref name="a"/> and <paramref name="b"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The sum of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Add(4.5, 5.4);
        /// if (c > 10)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.OverflowException">
        /// Thrown when one parameter is max and the other
        /// is greater than 0.</exception>
        /// See <see cref="Math.Add(int, int)"/> to add integers.
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
```

```
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0)
                || (b == double.MaxValue && a > 0))
            {
                throw new System.OverflowException();
            }

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <summary>
        /// Subtracts <paramref name="b"/> from <paramref name="a"/>
        /// and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Subtract(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Subtract(double, double)"/> to subtract
doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <summary>
        /// Subtracts a double <paramref name="b"/> from another
        /// double <paramref name="a"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The difference between two doubles.
        /// </returns>
        /// <example>
        /// <code>
```

```
        /// double c = Math.Subtract(4.5, 5.4);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Subtract(int, int)"/> to subtract integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <summary>
        /// Multiplies two integers <paramref name="a"/>
        /// and <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Multiply(4, 5);
        /// if (c > 100)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(double, double)"/> to multiply
doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Divide(int, int)"/>
        /// <param name="a">An integer.</param>
        /// <param name="b">An integer.</param>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
        /// <summary>
        /// Multiplies two doubles <paramref name="a"/> and
        /// <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The product of two doubles.
        /// </returns>
```

```
        /// <example>
        /// <code>
        /// double c = Math.Multiply(4.5, 5.4);
        /// if (c > 100.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// See <see cref="Math.Multiply(int, int)"/> to multiply integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Divide(double, double)"/>
        /// <param name="a">A double precision number.</param>
        /// <param name="b">A double precision number.</param>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }

        // Divides an integer by another and returns the result
        /// <summary>
        /// Divides an integer <paramref name="a"/> by another
        /// integer <paramref name="b"/> and returns the result.
        /// </summary>
        /// <returns>
        /// The quotient of two integers.
        /// </returns>
        /// <example>
        /// <code>
        /// int c = Math.Divide(4, 5);
        /// if (c > 1)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(double, double)"/> to divide doubles.
        /// <seealso cref="Math.Add(int, int)"/>
        /// <seealso cref="Math.Subtract(int, int)"/>
        /// <seealso cref="Math.Multiply(int, int)"/>
        /// <param name="a">An integer dividend.</param>
        /// <param name="b">An integer divisor.</param>
        public static int Divide(int a, int b)
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <summary>
        /// Divides a double <paramref name="a"/> by another double
        /// <paramref name="b"/> and returns the result.
```

```
        /// </summary>
        /// <returns>
        /// The quotient of two doubles.
        /// </returns>
        /// <example>
        /// <code>
        /// double c = Math.Divide(4.5, 5.4);
        /// if (c > 1.0)
        /// {
        ///     Console.WriteLine(c);
        /// }
        /// </code>
        /// </example>
        /// <exception cref="System.DivideByZeroException">
        /// Thrown when <paramref name="b"/> is equal to 0.
        /// </exception>
        /// See <see cref="Math.Divide(int, int)"/> to divide integers.
        /// <seealso cref="Math.Add(double, double)"/>
        /// <seealso cref="Math.Subtract(double, double)"/>
        /// <seealso cref="Math.Multiply(double, double)"/>
        /// <param name="a">A double precision dividend.</param>
        /// <param name="b">A double precision divisor.</param>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

You may find that the code is obscured by all the comments. The final example shows how you would adapt this library to use the `include` tag. You move all the documentation to an XML file:

XML

```
<docs>
  <members name="math">
   <Math>
    <summary>
      The main <c>Math</c> class.
      Contains all methods for performing basic math functions.
      </summary>
      <remarks>
      <para>This class can add, subtract, multiply and divide.</para>
      <para>These operations can be performed on both integers and doubles.
</para>
      </remarks>
    </Math>
    <AddInt>
      <summary>
      Adds two integers <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
```

```
      </summary>
      <returns>
      The sum of two integers.
      </returns>
      <example>
      <code>
      int c = Math.Add(4, 5);
      if (c > 10)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one
      parameter is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(double, double)"/> to add doubles.
      <seealso cref="Math.Subtract(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <seealso cref="Math.Divide(int, int)"/>
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </AddInt>
    <AddDouble>
      <summary>
      Adds two doubles <paramref name="a"/> and <paramref name="b"/>
      and returns the result.
      </summary>
      <returns>
      The sum of two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Add(4.5, 5.4);
      if (c > 10)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.OverflowException">Thrown when one parameter
is max
      and the other is greater than 0.</exception>
      See <see cref="Math.Add(int, int)"/> to add integers.
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <seealso cref="Math.Divide(double, double)"/>
      <param name="a">A double precision number.</param>
      <param name="b">A double precision number.</param>
    </AddDouble>
    <SubtractInt>
      <summary>
      Subtracts <paramref name="b"/> from <paramref name="a"/> and
      returns the result.
      </summary>
```

```
      <returns>
      The difference between two integers.
      </returns>
      <example>
      <code>
      int c = Math.Subtract(4, 5);
      if (c > 1)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      See <see cref="Math.Subtract(double, double)"/> to subtract doubles.
      <seealso cref="Math.Add(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <seealso cref="Math.Divide(int, int)"/>
      <param name="a">An integer.</param>
      <param name="b">An integer.</param>
    </SubtractInt>
    <SubtractDouble>
      <summary>
      Subtracts a double <paramref name="b"/> from another
      double <paramref name="a"/> and returns the result.
      </summary>
      <returns>
      The difference between two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Subtract(4.5, 5.4);
      if (c > 1)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      See <see cref="Math.Subtract(int, int)"/> to subtract integers.
      <seealso cref="Math.Add(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <seealso cref="Math.Divide(double, double)"/>
      <param name="a">A double precision number.</param>
      <param name="b">A double precision number.</param>
    </SubtractDouble>
    <MultiplyInt>
      <summary>
      Multiplies two integers <paramref name="a"/> and
      <paramref name="b"/> and returns the result.
      </summary>
      <returns>
      The product of two integers.
      </returns>
      <example>
      <code>
      int c = Math.Multiply(4, 5);
      if (c > 100)
```

```
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(double, double)"/> to multiply doubles.
    <seealso cref="Math.Add(int, int)"/>
    <seealso cref="Math.Subtract(int, int)"/>
    <seealso cref="Math.Divide(int, int)"/>
    <param name="a">An integer.</param>
    <param name="b">An integer.</param>
  </MultiplyInt>
  <MultiplyDouble>
    <summary>
    Multiplies two doubles <paramref name="a"/> and
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The product of two doubles.
    </returns>
    <example>
    <code>
    double c = Math.Multiply(4.5, 5.4);
    if (c > 100.0)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    See <see cref="Math.Multiply(int, int)"/> to multiply integers.
    <seealso cref="Math.Add(double, double)"/>
    <seealso cref="Math.Subtract(double, double)"/>
    <seealso cref="Math.Divide(double, double)"/>
    <param name="a">A double precision number.</param>
    <param name="b">A double precision number.</param>
  </MultiplyDouble>
  <DivideInt>
    <summary>
    Divides an integer <paramref name="a"/> by another integer
    <paramref name="b"/> and returns the result.
    </summary>
    <returns>
    The quotient of two integers.
    </returns>
    <example>
    <code>
    int c = Math.Divide(4, 5);
    if (c > 1)
    {
      Console.WriteLine(c);
    }
    </code>
    </example>
    <exception cref="System.DivideByZeroException">
    Thrown when <paramref name="b"/> is equal to 0.
```

```
      </exception>
      See <see cref="Math.Divide(double, double)"/> to divide doubles.
      <seealso cref="Math.Add(int, int)"/>
      <seealso cref="Math.Subtract(int, int)"/>
      <seealso cref="Math.Multiply(int, int)"/>
      <param name="a">An integer dividend.</param>
      <param name="b">An integer divisor.</param>
    </DivideInt>
    <DivideDouble>
      <summary>
      Divides a double <paramref name="a"/> by another
      double <paramref name="b"/> and returns the result.
      </summary>
      <returns>
      The quotient of two doubles.
      </returns>
      <example>
      <code>
      double c = Math.Divide(4.5, 5.4);
      if (c > 1.0)
      {
        Console.WriteLine(c);
      }
      </code>
      </example>
      <exception cref="System.DivideByZeroException">Thrown when <paramref
name="b"/> is equal to 0.</exception>
      See <see cref="Math.Divide(int, int)"/> to divide integers.
      <seealso cref="Math.Add(double, double)"/>
      <seealso cref="Math.Subtract(double, double)"/>
      <seealso cref="Math.Multiply(double, double)"/>
      <param name="a">A double precision dividend.</param>
      <param name="b">A double precision divisor.</param>
    </DivideDouble>
  </members>
</docs>
```

In the above XML, each member's documentation comments appear directly inside a tag named after what they do. You can choose your own strategy. The code uses the `<include>` tag to reference the appropriate element in the XML file:

C#

```
namespace IncludeTag
{

    /*
        The main Math class
        Contains all methods for performing basic math functions
    */
    /// <include file='include.xml'
path='docs/members[@name="math"]/Math/*'/>
```

```
    public class Math
    {
        // Adds two integers and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/AddInt/*'/>
        public static int Add(int a, int b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == int.MaxValue && b > 0) || (b == int.MaxValue && a >
0))
                throw new System.OverflowException();

            return a + b;
        }

        // Adds two doubles and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/AddDouble/*'/>
        public static double Add(double a, double b)
        {
            // If any parameter is equal to the max value of an integer
            // and the other is greater than zero
            if ((a == double.MaxValue && b > 0) || (b == double.MaxValue &&
a > 0))
                throw new System.OverflowException();

            return a + b;
        }

        // Subtracts an integer from another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/SubtractInt/*'/>
        public static int Subtract(int a, int b)
        {
            return a - b;
        }

        // Subtracts a double from another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/SubtractDouble/*'/>
        public static double Subtract(double a, double b)
        {
            return a - b;
        }

        // Multiplies two integers and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyInt/*'/>
        public static int Multiply(int a, int b)
        {
            return a * b;
        }

        // Multiplies two doubles and returns the result
```

```
        /// <include file='include.xml'
path='docs/members[@name="math"]/MultiplyDouble/*'/>
        public static double Multiply(double a, double b)
        {
            return a * b;
        }

        // Divides an integer by another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/DivideInt/*'/>
        public static int Divide(int a, int b)
        {
            return a / b;
        }

        // Divides a double by another and returns the result
        /// <include file='include.xml'
path='docs/members[@name="math"]/DivideDouble/*'/>
        public static double Divide(double a, double b)
        {
            return a / b;
        }
    }
}
```

- The `file` attribute represents the name of the XML file containing the documentation.
- The `path` attribute represents an XPath query to the `tag name` present in the specified `file`.
- The `name` attribute represents the name specifier in the tag that precedes the comments.
- The `id` attribute, which can be used in place of `name`, represents the ID for the tag that precedes the comments.

C# Compiler Errors

Article • 05/21/2024

Some C# compiler errors have corresponding topics that explain why the error is generated, and, in some cases, how to fix the error. Use one of the following steps to see whether help is available for a particular error message.

- If you're using Visual Studio, choose the error number (for example, CS0029) in the Output Window, and then choose the F1 key.
- Type the error number in the *Filter by title* box in the table of contents.

If none of these steps leads to information about your error, go to the end of this page, and send feedback that includes the number or text of the error.

For information about how to configure error and warning options in C#, see C# compiler options or the Visual Studio Build Page, Project Designer (C#).

> **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.

See also

- C# Compiler Options
- Build Page, Project Designer (C#)
- **WarningLevel** (C# Compiler Options)
- **NoWarn** (C# Compiler Options)

Preprocessor errors and warnings

10/09/2025

The compiler generates the following errors for incorrect use of preprocessor directives:

- CS1024: *Preprocessor directive expected*
- CS1025: *Single-line comment or end-of-line expected*
- CS1027: *#endif directive expected*
- CS1028: *Unexpected preprocessor directive*
- CS1029: *#error: 'text'*
- CS1030: *#warning: 'text'*
- CS1032: *Cannot define/undefine preprocessor symbols after first token in file*
- CS1038: *#endregion directive expected*
- CS1040: *Preprocessor directives must appear as the first non-white-space character on a line*
- CS1517: *Invalid preprocessor expression*
- CS1560: *Invalid filename specified for preprocessor directive. Filename is too long or not a valid filename*
- CS1576: *The line number specified for #line directive is missing or invalid*
- CS1578: *Filename, single-line comment or end-of-line expected*
- CS1633: *Unrecognized #pragma directive*
- CS1634: *Expected disable or restore*
- CS1635: *Cannot restore warning 'warning code' because it was disabled globally*
- CS1691: *'number' is not a valid warning number*
- CS1692: *Invalid number*
- CS1694: *Invalid filename specified for preprocessor directive. Filename is too long or not a valid filename*
- CS1695: *Invalid #pragma checksum syntax; should be #pragma checksum "filename" "{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}" "XXXX..."*
- CS1696: *Single-line comment or end-of-line expected*
- CS1709: *Filename specified for preprocessor directive is empty*
- CS7009: *Cannot use #r after first token in file*
- CS7010: *Quoted file name expected*
- CS7011: *#r is only allowed in scripts*
- CS8097: *#load is only allowed in scripts*
- CS8098: *Cannot use #load after first token in file*
- CS8938: *The #line directive value is missing or out of range*
- CS8939: *The #line directive end position must be greater than or equal to the start position*
- CS8996: *Raw string literals are not allowed in preprocessor directives*

- CS9028: *The #line span directive requires space before the first parenthesis, before the character offset, and before the file name*
- CS9297: `#:` *directives cannot be after first token in file*
- CS9298: `#:` *directives can be only used in file-based programs (*`-features:FileBasedProgram`*)*
- CS9299: `#:` *directives cannot be after* `#if` *directive*
- CS9314: `#!` *directives can be only used in scripts or file-based programs*

Invalid preprocessor directive syntax

- **CS1024**: *Preprocessor directive expected*
- **CS1025**: *Single-line comment or end-of-line expected*
- **CS1027**: *#endif directive expected*
- **CS1028**: *Unexpected preprocessor directive*
- **CS1038**: *#endregion directive expected*
- **CS1040**: *Preprocessor directives must appear as the first non-white-space character on a line*
- **CS1517**: *Invalid preprocessor expression*
- **CS1633**: *Unrecognized #pragma directive*
- **CS1696**: *Single-line comment or end-of-line expected*
- **CS8996**: *Raw string literals are not allowed in preprocessor directives*

These errors indicate that you used invalid syntax for preprocessor directives. Common causes include:

- Using an unrecognized directive after `#` (CS1024, CS1633).
- Including multiline comments on directive lines (CS1025, CS1696).
- Using directives in unexpected locations (CS1028).
- Missing required matching directives (CS1027, CS1038).
- Not defining or undefining symbols as the first token in the file (CS1032)
- Not placing the directive as the first token on a line (CS1040).
- Using invalid expressions in conditional compilation (CS1517).
- Using raw string literals in preprocessor directives (CS8996).

CS1024 example - Preprocessor directive expected:

```
C#
```

```
#import System   // CS1024 - "import" is not a valid directive
```

CS1025 example - Single-line comment or end-of-line expected:

C#

```
#if true /* hello
*/   // CS1025 - multiline comment not allowed
#endif
```

CS1027 example - #endif directive expected:

C#

```
#if true   // CS1027 - missing #endif
class Test { }
```

CS1028 example - Unexpected preprocessor directive:

C#

```
#endif   // CS1028 - no matching #if
```

CS1032 example - `#define` and `#undef` preprocessor directives must appear before other tokens:

C#

```
/* Comment */
#define X   // CS1032 - directive not first token in file
```

CS1038 example - #endregion directive expected:

C#

```
#region testing
class Test { }
// CS1038 - missing #endregion
```

CS1040 example - Preprocessor directives must appear as the first non-white-space character:

C#

```
/* Comment */ #define X   // CS1040 - directive not first on line
```

CS1517 example - Invalid preprocessor expression:

C#

```
C#

#if 1           // CS1517 - numeric literals not allowed
#endif
#if ~symbol     // CS1517 - bitwise operators not allowed
#endif
```

CS1633 example - Unrecognized #pragma directive:

```
C#

#pragma unknown  // CS1633 - "unknown" is not a valid pragma
```

CS8996 example - Raw string literals are not allowed in preprocessor directives:

```
C#

// CS8996.cs
#pragma checksum """raw_string""" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "hash"
// CS8996
class Test { }
```

To fix this error, use a regular string literal:

```
C#

#pragma checksum "filename.cs" "{406EA660-64CF-4C82-B6F0-42D48172A799}" "hash"  //
OK
class Test { }
```

To fix these errors, ensure your preprocessor directives follow the correct syntax rules described in the preprocessor directives documentation.

#error and #warning directive errors

- **CS1029**: *#error: 'text'*
- **CS1030**: *#warning: 'text'*
- **CS1634**: *Expected disable or restore*
- **CS1635**: *Cannot restore warning 'warning code' because it was disabled globally*
- **CS1691**: *'number' is not a valid warning number*
- **CS1692**: *Invalid number*

These errors occur when the compiler processes #error, #warning, and #pragma warning directives. These directives allow you to generate custom error and warning messages during

compilation and control warning behavior.

CS1029 displays the text of an error defined with the `#error` directive:

C#

```
// CS1029.cs
class Sample
{
   static void Main()
   {
      #error Let's give an error here   // CS1029
   }
}
```

Compilation produces the following output:

Console

```
example.cs(9,8): error CS1029: #error: 'Let's give an error here   // CS1029  '
```

CS1030 displays the text of a warning defined with the `#warning` directive:

C#

```
// CS1030.cs
class Sample
{
   static void Main()
   {
      #warning Let's give a warning here
   }
}
```

Compilation produces the following output:

Console

```
example.cs(6,16): warning CS1030: #warning: 'Let's give a warning here'
```

CS1634 occurs when a `#pragma warning` clause is badly formed, such as when disable or restore is omitted:

C#

```
// CS1634.cs
// compile with: /W:1
```

```
#pragma warning   // CS1634
// Try this instead:
// #pragma warning disable 0219

class MyClass
{
  public static void Main()
  {
  }
}
```

CS1635 occurs when you use the `/nowarn` command line option or project setting to disable a warning globally, but use `#pragma warning restore` to attempt to restore that warning:

C#

```
// CS1635.cs
// compile with: /w:1 /nowarn:162

enum MyEnum {one=1,two=2,three=3};

class MyClass
{
    public static void Main()
    {
#pragma warning disable 162

    if (MyEnum.three == MyEnum.two)
        System.Console.WriteLine("Duplicate");

#pragma warning restore 162  // CS1635
    }
}
```

CS1691 occurs when a number passed to the `#pragma warning` directive is not a valid warning number:

C#

```
// CS1691.cs
public class C
{
    int i = 1;
    public static void Main()
    {
        C myC = new C();
#pragma warning disable 151  // CS1691
// Try the following line instead:
// #pragma warning disable 1645
        myC.i++;
```

```
#pragma warning restore 151  // CS1691
// Try the following line instead:
// #pragma warning restore 1645
    }
}
```

CS1692 occurs when a number in preprocessor directives like `#pragma` and `#line` is invalid because it's too large, in the wrong format, or contains illegal characters:

C#

```
// CS1692.cs

#pragma warning disable a  // CS1692
// Try this instead:
// #pragma warning disable 1691

class A
{
    static void Main()
    {
    }
}
```

These directives are useful for conditional compilation scenarios where you want to alert developers about specific conditions in the code, or control which warnings are displayed during compilation. For more information about using these directives, see the preprocessor directives documentation and #pragma warning.

`#line` and file directive errors

- **CS1560**: *Invalid filename specified for preprocessor directive. Filename is too long or not a valid filename*
- **CS1576**: *The line number specified for #line directive is missing or invalid*
- **CS1578**: *Filename, single-line comment or end-of-line expected*
- **CS1694**: *Invalid filename specified for preprocessor directive. Filename is too long or not a valid filename*
- **CS1695**: *Invalid #pragma checksum syntax; should be #pragma checksum "filename" "{XXXXXXXX-XXXX-XXXX-XXXX-XXXXXXXXXXXX}" "XXXX..."*
- **CS1709**: *Filename specified for preprocessor directive is empty*
- **CS8938**: *The #line directive value is missing or out of range*
- **CS8939**: *The #line directive end position must be greater than or equal to the start position*
- **CS9028**: *The #line span directive requires space before the first parenthesis, before the character offset, and before the file name*

These errors indicate incorrect usage of the #line directive or file-related preprocessor directives. Common causes include:

- Invalid or missing filenames (CS1560, CS1694, CS1709).
- Incorrect line number format or values (CS1576, CS8938, CS8939).
- Missing proper syntax for filenames and comments (CS1578).
- Malformed `#pragma checksum` syntax (CS1695).
- Improper spacing in `#line` span directives (CS9028).

CS1560/CS1694 example - Invalid filename specified:

C#

```
#line 100
"MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890M
yFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyF
ile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFile1234567890MyFil
e1234567890.txt"   // CS1560/CS1694 - filename too long
```

CS1576 example - Line number missing or invalid:

C#

```
#line "abc.sc"         // CS1576 - missing line number
#line abc "file.cs"    // CS1576 - invalid line number format
```

CS1578 example - Filename, comment or end-of-line expected:

C#

```
#line 101 abc.cs   // CS1578 - filename not quoted
```

CS1695 example - Invalid #pragma checksum syntax:

C#

```
#pragma checksum "12345"  // CS1695 - missing GUID and checksum
```

CS1709 example - Empty filename:

C#

```
#pragma checksum "" "{406EA660-64CF-4C82-B6F0-42D48172A799}" ""  // CS1709 - empty
filename
```

CS8938 example - #line directive value missing or out of range:

```
C#

#line   // CS8938 - missing value
#line 0  // CS8938 - out of range (must be 1-16,707,566)
```

CS8939 example - #line end position error:

```
C#

#line (1, 10) - (1, 5) "file.cs"  // CS8939 - end column < start column
```

CS9028 example - #line span directive spacing:

```
C#

#line(1, 1) - (1, 10)"file.cs"  // CS9028 - missing spaces
```

To fix these errors, ensure your `#line` directives and file-related preprocessor directives follow the correct syntax as described in the preprocessor directives documentation.

Incorrect use of file-based apps directives

- **CS9297**: `#:` *directives cannot be after first token in file*
- **CS9298**: `#:` *directives can be only used in file-based programs (*`-features:FileBasedProgram`*)*
- **CS9299**: `#:` *directives cannot be after* `#if` *directive*
- **CS9314**: `#!` *directives can be only used in scripts or file-based programs*

These errors indicate that you used the `#:` directives for a file-based app incorrectly. You can learn more about the syntax for these directives in the article on preprocessor directives in the section on file-based apps. Or, you can explore file based apps by following the tutorial on file-based apps.

File contains script.cs directives

The following errors indicate script.cs syntax in a compiled C# file:

- **CS7009**: *Cannot use #r after first token in file*
- **CS7010**: *Quoted file name expected*

- **CS7011**: *#r is only allowed in scripts*
- **CS8097**: *#load is only allowed in scripts*

These directives aren't supported in compiled C#. You must remove them, or use script.cs.

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

Resolve errors and warnings related to attribute declarations or attribute use in your code

This article covers the following compiler errors:

- CS0243: *The Conditional attribute is not valid on 'method' because it is an override method.*
- CS0404: *Attribute is not valid on this declaration type.*
- CS0415: *This attribute is valid only on an indexer that is not an explicit interface member declaration.*
- CS0416: *'type parameter': an attribute argument cannot use type parameters.*
- CS0447: *Attribute cannot be used with type arguments.*
- CS0577: *The Conditional attribute is not valid because it is a constructor, destructor, operator, lambda expression, or explicit interface implementation.*
- CS0578: *The Conditional attribute is not valid on 'function' because its return type is not void.*
- CS0582: *The Conditional attribute is not valid on interface members.*
- CS0609: *Cannot set the attribute on an indexer marked override.*
- CS0625: *Instance field in types marked with StructLayout(LayoutKind.Explicit) must have a FieldOffset attribute.*
- CS0629: *Conditional member 'member' cannot implement interface member 'base class member' in type 'Type Name'.*
- CS0636: *The FieldOffset attribute can only be placed on members of types marked with the StructLayout(LayoutKind.Explicit).*
- CS0637: *The FieldOffset attribute is not allowed on static or const fields.*
- CS0641: *This attribute is only valid on classes derived from System.Attribute`.*
- CS0646: *Cannot specify the DefaultMember attribute on a type containing an indexer.*
- CS0653: *Cannot apply attribute class 'class' because it is abstract.*
- CS0657: *Location is not a valid attribute location for this declaration. Valid attribute locations for this declaration are listed. All attributes in this block will be ignored.*
- CS0658: *Location is not a recognized attribute location. Valid attribute locations for this declaration are listed. All attributes in this block will be ignored.*
- CS0685: *Conditional member 'member' cannot have an out parameter.*
- CS7014: *Attributes are not valid in this context.*
- CS7046: *Attribute parameter must be specified.*
- CS7047: *Attribute parameter 'parameter1' or 'parameter2' must be specified.*
- CS7067: *Attribute constructor parameter is optional, but no default parameter value was specified.*
- CS9331: *Attribute cannot be applied manually.*

Attribute arguments and parameters

The following errors occur when you use attribute arguments or parameters incorrectly:

- **CS0416**: *'type parameter': an attribute argument cannot use type parameters.*
- **CS0447**: *Attribute cannot be used with type arguments.*
- **CS7046**: *Attribute parameter must be specified.*
- **CS7047**: *Attribute parameter 'parameter1' or 'parameter2' must be specified.*
- **CS7067**: *Attribute constructor parameter is optional, but no default parameter value was specified.*

To correct these errors, follow these rules.

- Use compile-time constant values for attribute arguments instead of type parameters (**CS0416**). Attribute arguments must be evaluated at compile time.
- Don't use type arguments with attributes (**CS0447**). Type arguments aren't allowed in attribute usage.
- Provide all required attribute parameters when you apply the attribute (**CS7046**, **CS7047**). Check the attribute's constructor to see which parameters are mandatory.
- Specify default values for optional constructor parameters when you define custom attributes (**CS7067**). Use the syntax `parameterType parameterName = defaultValue` in the attribute constructor.

For more information, see Attributes and Generics.

> **Note**
>
> **CS0447** is no longer used in Roslyn.

Attribute class requirements

The following errors occur when you define attribute classes that don't meet the required constraints:

- **CS0404**: *Attribute is not valid on this declaration type.*
- **CS0641**: *This attribute is only valid on classes derived from System.Attribute`.*
- **CS0653**: *Cannot apply attribute class 'class' because it is abstract.*

To correct these errors, follow these rules.

- Apply attributes only to valid declaration types (**CS0404**). Check the attribute's `AttributeUsage` to see which targets are allowed.

- Remove the `abstract` modifier from attribute classes, or derive from a non-abstract attribute class (**CS0653**). Attributes must be instantiable.
- Apply AttributeUsageAttribute only to classes that derive from `Attribute` (**CS0641**). This attribute controls how other attributes are used.

For more information, see Create custom attributes.

Attribute location context

The following errors occur when you apply attributes in invalid locations or with incorrect target specifiers:

- **CS0657**: *Location is not a valid attribute location for this declaration. Valid attribute locations for this declaration are listed. All attributes in this block will be ignored.*
- **CS0658**: *Location is not a recognized attribute location. Valid attribute locations for this declaration are listed. All attributes in this block will be ignored.*
- **CS7014**: *Attributes are not valid in this context.*

To correct these errors, follow these rules. For more information, see Attribute Targets.

- Use valid attribute target specifiers for the declaration (**CS0657**). Check the error message to see which targets are valid for the specific declaration.
- Remove invalid attribute target specifiers (**CS0658**). Valid specifiers include `assembly:`, `module:`, `type:`, `method:`, `property:`, `field:`, `event:`, `param:`, and `return:`.
- Move attributes to valid contexts (**CS7014**). Attributes can only be applied to program elements that support them, as defined by AttributeUsageAttribute.
- Apply attributes at the appropriate scope level. Use `assembly:` or `module:` prefixes for assembly-level or module-level attributes.

Predefined attributes

The following errors occur when you use specific predefined .NET attributes incorrectly:

- **CS0415**: *This attribute is valid only on an indexer that is not an explicit interface member declaration.*
- **CS0609**: *Cannot set the attribute on an indexer marked override.*
- **CS0625**: *Instance field in types marked with StructLayout(LayoutKind.Explicit) must have a FieldOffset attribute.*
- **CS0636**: *The FieldOffset attribute can only be placed on members of types marked with the StructLayout(LayoutKind.Explicit).*
- **CS0637**: *The FieldOffset attribute is not allowed on static or const fields.*

- **CS0646**: *Cannot specify the DefaultMember attribute on a type containing an indexer.*
- **CS9331**: *Attribute cannot be applied manually.*

To correct these errors, follow these rules. For more information, see Indexers and Platform Invoke (P/Invoke).

- Remove IndexerNameAttribute from explicit interface implementations (**CS0415**). Apply it only to public indexers.
- Remove `IndexerName` from indexers marked with `override` (**CS0609**). Override indexers inherit the name from the base class.
- Add FieldOffsetAttribute to all instance fields in types with StructLayoutAttribute set to `LayoutKind.Explicit` (**CS0625**). Explicit layout requires explicit field positions.
- Apply `FieldOffset` only to types with `StructLayout(LayoutKind.Explicit)` (**CS0636**). Add the `StructLayout` attribute to the type declaration.
- Remove `FieldOffset` from `static` or `const` fields (**CS0637**). Explicit layout applies only to instance fields.
- Remove DefaultMemberAttribute from types that contain indexers (**CS0646**). Indexers automatically define the default member.
- Replace compiler-generated attributes with the equivalent C# syntax (**CS9331**). Use language keywords instead of manually applying reserved attributes.

Conditional attribute usage

The following errors occur when you apply the ConditionalAttribute in ways that violate its usage restrictions:

- **CS0243**: *The Conditional attribute is not valid on 'method' because it is an override method.*
- **CS0577**: *The Conditional attribute is not valid because it is a constructor, destructor, operator, lambda expression, or explicit interface implementation.*
- **CS0578**: *The Conditional attribute is not valid on 'function' because its return type is not void.*
- **CS0582**: *The Conditional attribute is not valid on interface members.*
- **CS0629**: *Conditional member 'member' cannot implement interface member 'base class member' in type 'Type Name'.*
- **CS0685**: *Conditional member 'member' cannot have an out parameter.*

To correct these errors, follow these rules. For more information, see ConditionalAttribute and Attributes.

- Remove the `Conditional` attribute from override methods, or remove the `override` keyword (**CS0243**). The compiler binds to the base method, not the override.

- Don't apply `Conditional` to constructors, finalizers, operators, lambda expressions, or explicit interface implementations (**CS0577**). These member types don't support conditional compilation.
- Change the method's return type to `void`, or remove the `Conditional` attribute (**CS0578**). Conditional methods must return `void` because calls may be compiled out.
- Remove the `Conditional` attribute from interface member declarations (**CS0582**). Interface members can't be conditional.
- Remove the `Conditional` attribute from interface member implementations (**CS0629**). Methods implementing interface members can't be conditional.
- Remove `out` parameters from conditional methods, or remove the `Conditional` attribute (**CS0685**). The `out` variable value would be undefined when the method call is compiled out.

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

---

Last updated on 11/12/2025

Resolve warnings related to language features and versions

05/31/2025

This article covers the following compiler warnings:

- **CS8022, CS8023, CS8024, CS8025, CS8026, CS8059, CS8107, CS8302, CS8320, CS8370, CS8400, CS8773, CS8936, CS9058**: *Feature is not available. Use newer language version.*
- **CS8058**: *Feature is experimental.*
- **CS8192**: *Provided language version is unsupported or invalid*
- **CS8303**: *Specified language version cannot have leading zeroes*
- **CS8304**: *Compiler version is less than language version*
- **CS1738**: *Named argument specifications must appear after all fixed arguments have been specified.*
- **CS8306**: *Tuple element name is inferred.*
- **CS8314**: *An expression of type cannot be handled by a pattern of type*
- **CS8371**: *Field-targeted attributes on auto-properties are not supported in language version*
- **CS8401**: *To use `@$` instead of `$@` for an interpolated verbatim string, use newer language version.*
- **CS8511**: *An expression of type cannot be handled by a pattern of type.*
- **CS8627**: *A nullable type parameter must be known to be a value type or non-nullable reference type*
- **CS8630**: *Invalid nullable options. Use newer language version*
- **CS8652**: *The modifier is not valid for this item.*
- **CS8704**: *Type does not implement interface member. It cannot implicitly implement a non-public member.*
- **CS8706**: *Type cannot implement interface member because a feature is not available in this version.*
- **CS8904**: *Invalid variance: The type parameter must be valid.*
- **CS8912**: *Inheriting from a record with a sealed 'Object.ToString' is not supported.*
- **CS8919**: *Cannot implement specified interface member in type because the target runtime doesn't support static abstract members in interfaces*
- **CS8929**: *Method cannot implement interface member in type because the target runtime doesn't support static abstract members in interfaces.*
- **CS8957**: *Conditional expression is not valid in language version because a common type was not found between types.*
- **CS8967**: *Newlines inside a non-verbatim interpolated string are not supported in C#*
- **CS9014**: *Error: Use of possibly unassigned property. Upgrade to auto-default the property.*
- **CS9015**: *Error: Use of possibly unassigned field. Upgrade to auto-default the field.*

- **CS9016**: *Warning: Use of possibly unassigned property. Upgrade to auto-default the property.*
- **CS9017**: *Warning: Use of possibly unassigned field. Upgrade to auto-default the field.*
- **CS9064**: *Target runtime doesn't support ref fields.*
- **CS9103**: *Definition in a module with an unrecognized RefSafetyRulesAttribute version, expecting '11'.*
- **CS9171**: *Target runtime doesn't support inline array types.*
- **CS9194**: *Argument may not be passed with the `ref` keyword. To pass `ref` arguments to `in` parameters, upgrade to language version 12 or greater.*
- **CS9202**: *Feature is not available in C# 12.0. Please use newer language version*
- **CS9211**: *The diagnosticId argument to the 'Experimental' attribute must be a valid identifier.*
- **CS9240**: *Target runtime doesn't support by-ref-like generics.*
- **CS9260**: *Feature is not available in C# 13.0. Please use newer language version**
- **CS9268**: *Type is for evaluation purposes only and is subject to change or removal in future updates. Suppress this diagnostic to proceed.*
- **CS9269**: *UnscopedRefAttribute is only valid in C# 11 or later or when targeting net7.0 or later.*
- CS9271: *The type '`Microsoft.CodeAnalysis.EmbeddedAttribute`' must be non-generic, internal, sealed, non-static, have a parameterless constructor, inherit from System.Attribute, and be able to be applied to any type.*

In addition, the following errors and warnings relate to struct initialization changes in recent versions:

- CS0171, CS8881: *Backing field for automatically implemented property 'name' must be fully assigned before control is returned to the caller.*
- CS0188, CS8885: *The 'this' object cannot be used before all of its fields are assigned to*
- CS0843, CS8880: *Backing field for automatically implemented property 'name' must be fully assigned before control is returned to the caller*
- CS8305: *Feature is for evaluation purposes only and is subject to change or removal in future updates.*
- CS9204: *Type is for evaluation purposes only and is subject to change or removal in future updates. Suppress this diagnostic to proceed.*

The cause behind all these errors and warnings is that the compiler installed supports a newer version of C# than the version your project has selected. The C# compiler can conform to any previous version. You can validate syntax against an earlier version of C#, or because your project must support older libraries or runtimes.

There are two possible causes and three ways to address these errors and warnings.

Update your target framework

The compiler determines a default based on these rules:

Expand table

Target	Version	C# language version default
.NET	10.x	C# 14
.NET	9.x	C# 13
.NET	8.x	C# 12
.NET	7.x	C# 11
.NET	6.x	C# 10
.NET	5.x	C# 9.0
.NET Core	3.x	C# 8.0
.NET Core	2.x	C# 7.3
.NET Standard	2.1	C# 8.0
.NET Standard	2.0	C# 7.3
.NET Standard	1.x	C# 7.3
.NET Framework	all	C# 7.3

If your selected framework doesn't match the language version required, you can upgrade the target framework.

Select the matching language version

You might have an older target framework selected in your project file. If you remove the `LangVersion` element from your project file, the compiler uses the default value listed in the preceding section. The following table shows all current C# language versions. You can also specify a specific language version to enable newer features.

Expand table

Value	Meaning
`preview`	The compiler accepts all valid language syntax from the latest preview version.

Value	Meaning
`latest`	The compiler accepts syntax from the latest released version of the compiler (including minor version).
`latestMajor` or `default`	The compiler accepts syntax from the latest released major version of the compiler.
`14.0`	The compiler accepts only syntax that is included in C# 14 or lower.
`13.0`	The compiler accepts only syntax that is included in C# 13 or lower.
`12.0`	The compiler accepts only syntax that is included in C# 12 or lower.
`11.0`	The compiler accepts only syntax that is included in C# 11 or lower.
`10.0`	The compiler accepts only syntax that is included in C# 10 or lower.
`9.0`	The compiler accepts only syntax that is included in C# 9 or lower.
`8.0`	The compiler accepts only syntax that is included in C# 8.0 or lower.
`7.3`	The compiler accepts only syntax that is included in C# 7.3 or lower.
`7.2`	The compiler accepts only syntax that is included in C# 7.2 or lower.
`7.1`	The compiler accepts only syntax that is included in C# 7.1 or lower.
`7`	The compiler accepts only syntax that is included in C# 7.0 or lower.
`6`	The compiler accepts only syntax that is included in C# 6.0 or lower.
`5`	The compiler accepts only syntax that is included in C# 5.0 or lower.
`4`	The compiler accepts only syntax that is included in C# 4.0 or lower.
`3`	The compiler accepts only syntax that is included in C# 3.0 or lower.
`ISO-2` or `2`	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
`ISO-1` or `1`	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

You can learn more about the language versions supported for each framework version in the article on Configure language version in the language reference section.

Avoid the updated feature

If you must support older libraries or runtimes, you might need to avoid using newer features.

Enable experimental features

The diagnostics for experimental features can be disabled to use the experimental feature.

> ⚠ **Warning**
>
> Experimental features are subject to changes. The APIs may change, or they may be removed in future updates. Including experimental features is a way for library authors to get feedback on ideas and concepts for future development. Use extreme caution when using any feature marked as experimental.

You can also declare your own experimental features using the System.Diagnostics.CodeAnalysis.ExperimentalAttribute. The compiler emits *CS9211* if the identifier used for the experimental feature is not a valid identifier.

Breaking changes on struct initialization

All these errors and warnings help ensure that `struct` types are properly initialized before their fields are accessed. In earlier versions of C#, you must explicitly assign all fields in a struct in any constructor. The parameterless constructor initializes all fields to their default value. In later versions, all constructors initialize all fields. Either the field is explicitly set, set in a field initializer, or set to its default value.

- **CS0171**, **CS8881**: *Backing field for automatically implemented property 'name' must be fully assigned before control is returned to the caller.*
- **CS0188**, **CS8885**: *The 'this' object cannot be used before all of its fields are assigned to*
- **CS0843**, **CS8880**: *Backing field for automatically implemented property 'name' must be fully assigned before control is returned to the caller*

You can address this error by upgrading your language version to C# 11, every `struct` constructor initializes all fields. If that's not a possible option, you must explicitly call the default constructor, as shown in the following example:

C#

```
struct S
{
    public int AIProp { get; set; }
    public S(int i){} //CS0843
    // Try the following lines instead.
    // public S(int i) : this()
    // {
    //     AIProp = i;
```

```
    // }
}

class Test
{
    static int Main()
    {
        return 1;
    }
}
```

Implementation specific attributes

- **CS9271**: *The type '`Microsoft.CodeAnalysis.EmbeddedAttribute`' must be non-generic, internal, sealed, non-static, have a parameterless constructor, inherit from System.Attribute, and be able to be applied to any type.*

The compiler generates the source for this attribute when needed for your source code. You shouldn't declare this type.

Resolve errors and warnings related to assembly references

08/15/2025

- CS0012: *The type 'type' is defined in an assembly that is not referenced. You must add a reference to assembly 'assembly'.*
- CS0234: *The type or namespace name does not exist in the namespace (are you missing an assembly reference?)*
- CS0246: *The type or namespace name could not be found (are you missing a using directive or an assembly reference?)*
- CS0400: *The type or namespace name could not be found in the global namespace (are you missing an assembly reference?)*
- CS0735: *Invalid type specified as an argument for TypeForwardedToAttribute attribute.*
- CS1068: *The type name could not be found in the global namespace. This type has been forwarded to another assembly. Consider adding a reference to that assembly.*
- CS1069: *The type name could not be found in the namespace. This type has been forwarded to another assembly. Consider adding a reference to that assembly.*
- CS1070: *The type name could not be found. This type has been forwarded to another assembly. Consider adding a reference to that assembly.*
- CS1704: *An assembly with the same simple name has already been imported. Try removing one of the references or sign them to enable side-by-side.*
- CS1714: *The base class or interface of this type could not be resolved or is invalid.*
- CS1760: *Multiple assemblies refer to the same metadata but only one is a linked reference (specified using /link option); consider removing one of the references.*
- CS7008: *The assembly name is reserved and cannot be used as a reference in an interactive session.*
- CS7068: *Reference to type claims it is defined in this assembly, but it is not defined in source or any added modules.*
- CS7069: *Reference to type claims it is defined in another assembly, but it could not be found.*
- CS7071: *Assembly reference is invalid and cannot be resolved.*
- CS7079: *The type is defined in a module that has not been added. You must add the module.*
- CS8090: *There is an error in a referenced assembly.*
- CS8203: *Invalid assembly name.*
- CS9286: *Type does not contain a definition and no accessible extension member for receiver type could be found (are you missing a using directive or an assembly reference?)*

In addition, the following warnings are covered in this article:

- CS1683: *Reference to type 'Type Name' claims it is defined in this assembly, but it is not defined in source or any added modules.*

Missing references

The following errors and warnings indicate that you're missing an assembly reference:

- **CS0012**: *The type 'type' is defined in an assembly that is not referenced. You must add a reference to assembly 'assembly'.*
- **CS0234**: *The type or namespace name does not exist in the namespace (are you missing an assembly reference?)*
- **CS0246**: *The type or namespace name could not be found (are you missing a using directive or an assembly reference?)*
- **CS9286**: *Type does not contain a definition and no accessible extension member for receiver type could be found (are you missing a using directive or an assembly reference?)*

These compiler errors indicate one of these problems in your code:

- The project doesn't reference the required assembly. To fix this error, add a reference to the required assembly.
- You misspelled the name of a type. Check the name of the type.
- You used a variable name where the name of a System.Type was expected, such as in the typeof operator or the is operator.
- You used the global scope operator, (::) when the type isn't in the global namespace.
- You're accessing an extension member and either the namespace isn't specified in a `using` directive, or you're not referencing the assembly that contains the extension.

When the assembly appears to be referenced

If the assembly appears to be referenced in your project but you still receive CS0012, try these troubleshooting steps:

- Restore packages: Run `dotnet restore` to ensure all package references are properly resolved, especially after installing or uninstalling NuGet packages.

- Clear the NuGet package cache and restore:

  Console

  ```
  dotnet nuget locals all --clear
  dotnet restore
  ```

- Check for version conflicts: Verify that all referenced assemblies use compatible versions. Look for binding redirect warnings in the build output.
- Clean the solution and rebuild to ensure no stale references remain:

```
Console

dotnet clean
dotnet build
```

- Verify package integrity: If the error occurred after package operations, ensure the package was installed correctly by removing and reinstalling it:

```
Console

dotnet remove package [PackageName]
dotnet add package [PackageName]
```

Type forwarding

- **CS1068**: *The type name could not be found in the global namespace. This type has been forwarded to another assembly. Consider adding a reference to that assembly.*
- **CS1069**: *The type name could not be found in the namespace. This type has been forwarded to another assembly. Consider adding a reference to that assembly.*
- **CS1070**: *The type name could not be found. This type has been forwarded to another assembly. Consider adding a reference to that assembly.*

These errors indicate an error referencing a type forwarded to a different assembly. To address any of these errors, add a reference to the assembly indicated in the error message.

Duplicate references

The following errors indicate a duplicate assembly reference:

- **CS1704**: *An assembly with the same simple name has already been imported. Try removing one of the references or sign them to enable side-by-side.*
- **CS1760**: *Multiple assemblies refer to the same metadata but only one is a linked reference (specified using /link option); consider removing one of the references.*

To fix these errors, you must either remove one of the references, or resolve the duplication. Causes for duplication include:

- Multiple unsigned assemblies have the same name.

- Your project references multiple versions of the same assembly.

Invalid assembly reference

The following errors indicate that an assembly reference is invalid:

- **CS7008**: *The assembly name is reserved and cannot be used as a reference in an interactive session.*
- **CS7069**: *Reference to type claims it is defined in another assembly, but it could not be found.*
- **CS7071**: *Assembly reference is invalid and cannot be resolved.*
- **CS7079**: *The type is defined in a module that has not been added. You must add the module.*
- **CS8090**: *There is an error in a referenced assembly.*
- **CS8203**: *Invalid assembly name.*

The following warning also indicates an invalid reference assembly:

- **CS1683**: *Reference to type '*Type Name*' claims it is defined in this assembly, but it is not defined in source or any added modules*

Check that the assembly name is spelled correctly. The referenced assembly file might be invalid.

Resolve errors and warnings in constructor declarations

Article • 11/22/2024

This article covers the following compiler errors:

- CS0514: *static constructor cannot have an explicit 'this' or 'base' constructor call.*
- CS0515: *access modifiers are not allowed on static constructors.*
- CS0516: *Constructor 'constructor' can not call itself.*
- CS0517: *'class' has no base class and cannot call a base constructor.*
- CS0522: *structs cannot call base class constructors.*
- CS0526: *Interfaces cannot contain constructors.*
- CS0568: *Structs cannot contain explicit parameterless constructors.*
- CS0710: *Static classes cannot have instance constructors.*
- CS0768: *Constructor cannot call itself through another constructor.*
- CS8054: *Enums cannot contain explicit parameterless constructors.*
- CS8091: *cannot be extern and have a constructor initializer.*
- CS8861: *Unexpected argument list.*
- CS8862: *A constructor declared in a type with parameter list must have 'this' constructor initializer.*
- CS8358: *Cannot use attribute constructor because it has 'in' parameters.*
- CS8867: *No accessible copy constructor found in base type '{0}'.*
- CS8868: *A copy constructor in a record must call a copy constructor of the base, or a parameterless object constructor if the record inherits from object.*
- CS8878: *A copy constructor '{0}' must be public or protected because the record is not sealed.*
- CS8910: *The primary constructor conflicts with the synthesized copy constructor.*
- CS8958: *The parameterless struct constructor must be 'public'.*
- CS8982: *A constructor declared in a 'struct' with parameter list must have a 'this' initializer that calls the primary constructor or an explicitly declared constructor.*
- CS8983: *A 'struct' with field initializers must include an explicitly declared constructor.*
- CS9105: *Cannot use primary constructor parameter in this context.*
- CS9106: *Identifier is ambiguous between type and parameter in this context.* class as well.*
- CS9108: *Cannot use parameter that has ref-like type inside an anonymous method, lambda expression, query expression, or local function.*
- CS9109: *Cannot use* `ref`, `out`, *or* `in` *primary constructor parameter inside an instance member.*

- CS9110: *Cannot use primary constructor parameter that has ref-like type inside an instance member.*
- CS9111: *Anonymous methods, lambda expressions, query expressions, and local functions inside an instance member of a struct cannot access primary constructor parameter.*
- CS9112: *Anonymous methods, lambda expressions, query expressions, and local functions inside a struct cannot access primary constructor parameter also used inside an instance member.*
- CS9114: *A primary constructor parameter of a readonly type cannot be assigned to (except in init-only setter of the type or a variable initializer).*
- CS9115: *A primary constructor parameter of a readonly type cannot be returned by writable reference.*
- CS9116: *A primary constructor parameter of a readonly type cannot be used as a ref or out value (except in init-only setter of the type or a variable initializer).*
- CS9117: *Members of primary constructor parameter of a readonly type cannot be modified (except in init-only setter of the type or a variable initializer).*
- CS9118: *Members of primary constructor parameter of a readonly type cannot be returned by writable reference.*
- CS9119: *Members of primary constructor parameter of a readonly type cannot be used as a `ref` or `out` value (except in init-only setter of the type or a variable initializer).*
- CS9120: *Cannot return primary constructor parameter by reference.*
- CS9121: *Struct primary constructor parameter of type causes a cycle in the struct layout.*
- CS9122: *Unexpected parameter list.*
- CS9136: *Cannot use primary constructor parameter of type inside an instance member.*

In addition, the following warnings are covered in this article:

- CS0824: *Constructor 'name' is marked external.*
- CS9107: *Parameter is captured into the state of the enclosing type and its value is also passed to the base constructor. The value might be captured by the base class as well.*
- CS9113: *Parameter is unread.*
- CS9124: *Parameter is captured into the state of the enclosing type and its value is also used to initialize a field, property, or event.*
- CS9179: *Primary constructor parameter is shadowed by a member from base*

Static constructors

- **CS0514**: *Static constructor cannot have an explicit 'this' or 'base' constructor call.*
- **CS0515**: *Access modifiers are not allowed on static constructors.*

You can write at most one static constructor for a type. The declaration of a static constructor must obey the following rules:

- The static constructor has the `static` modifier, but no other modifiers, such as `public`, `protected`, `private`, or `internal`.
- The static constructor must be a parameterless constructor.
- The static constructor must not call `base()` or `this()`. If the base class includes a static constructor, the runtime calls it automatically.

Constructor declaration

- **CS0526**: *Interfaces cannot contain constructors.*
- **CS0710**: *Static classes cannot have instance constructors.*
- **CS8054**: *Enums cannot contain explicit parameterless constructors.*
- **CS8358**: *Cannot use attribute constructor because it has 'in' parameters.*
- **CS8091**: *A constructor cannot be extern and have a constructor initializer.*

Constructors are allowed only in `class` and `struct` types, including `record class` and `record struct` types. You can't define them in `enum` or `interface` types. Furthermore, attribute class types can't declare `in` parameters. Instead, pass parameters by value.

You can declare `extern` constructors, however you can't use `base()` or `this()` constructor calls to call another constructor from a constructor declared `extern`.

In addition, the following warnings can be generated for constructor declarations:

- **CS0824**: *Constructor is marked external.*

When a constructor is marked `extern`, the compiler can't guarantee the constructor exists. Therefore, the compiler generates this warning.

Constructors in struct types

- **CS0568**: *Structs cannot contain explicit parameterless constructors.*
- **CS8958**: *The parameterless struct constructor must be 'public'.*
- **CS8982**: *A constructor declared in a 'struct' with parameter list must have a 'this' initializer that calls the primary constructor or an explicitly declared constructor.*
- **CS8983**: *A 'struct' with field initializers must include an explicitly declared constructor.*

Recent features in C# remove earlier restrictions to `struct` types. **CS0568** is generated when you declare a parameterless instance constructor in older versions of C#. You can declare an explicit parameterless instance constructor in newer versions of C#. That explicit parameterless constructor must be `public`. If your `struct` declares any field initializers, you must also declare an explicit instance constructor. This constructor can be a parameterless constructor with an empty body.

When a `struct` type declares a primary constructor, including `record struct` types, all other instance constructors except a parameterless constructor must call the primary constructor or another explicitly declared constructor using `this()`.

Constructor calls with `base` and `this`

- **CS0516**: *Constructor can not call itself.*
- **CS0517**: *Class has no base class and cannot call a base constructor.*
- **CS0522**: *Structs cannot call base class constructors.*
- **CS0768**: *Constructor cannot call itself through another constructor.*

You can use `base()` and `this()` to have one constructor call another in the same type or the base type. Calling constructors can minimize duplicated constructor logic. You must follow these rules when calling another constructor using `this()` or `base()`:

- A constructor can't call itself either directly or indirectly through another constructor. For example, the following code is illegal:

```
C#
```

```
public class C
{
  public C() : this() // Error!
  {
  }
}

public class C2
{
  public class C2() : this(10) {}

  public class C2(int capacity) : this()
  {
      _capacity = capacity;
  }

  private int _capacity;
}
```

```
```

- Struct types can't call `base()`. Neither can System.Object.

Records and copy constructors

- **CS8867**: *No accessible copy constructor found in base type.*
- **CS8868**: *A copy constructor in a record must call a copy constructor of the base, or a parameterless object constructor if the record inherits from object.*
- **CS8878**: *A copy constructor must be public or protected because the record is not sealed.*
- **CS8910**: *The primary constructor conflicts with the synthesized copy constructor.*

Adding the `record` modifier to a `struct` or `class` type creates a record. Records include a compiler-synthesized copy constructor. You can write an explicit copy constructor yourself, but it must adhere to the following rules:

- Copy constructors must be `public` or `protected` unless the type is sealed.
- Copy constructors must call the `base()` copy constructor unless the base class is System.Object.
- Furthermore, the base type must have a copy constructor. `record` types always have a copy constructor.

Primary constructor declaration

The compiler emits the following errors when a primary constructor violates one or more rules on primary constructors for classes and structs:

- **CS8861**: *Unexpected argument list.*
- **CS8862**: *A constructor declared in a type with parameter list must have 'this' constructor initializer.*
- **CS9105**: *Cannot use primary constructor parameter in this context.*
- **CS9106**: *Identifier is ambiguous between type and parameter in this context.*
- **CS9108**: *Cannot use parameter that has ref-like type inside an anonymous method, lambda expression, query expression, or local function.*
- **CS9109**: *Cannot use `ref`, `out`, or `in` primary constructor parameter inside an instance member.*
- **CS9110**: *Cannot use primary constructor parameter that has ref-like type inside an instance member.*

- **CS9111**: *Anonymous methods, lambda expressions, query expressions, and local functions inside an instance member of a struct cannot access primary constructor parameter.*
- **CS9112**: *Anonymous methods, lambda expressions, query expressions, and local functions inside a struct cannot access primary constructor parameter also used inside an instance member.*
- **CS9114**: *A primary constructor parameter of a readonly type cannot be assigned to (except in init-only setter of the type or a variable initializer).*
- **CS9115**: *A primary constructor parameter of a readonly type cannot be returned by writable reference.*
- **CS9116**: *A primary constructor parameter of a readonly type cannot be used as a `ref` or `out` value (except in init-only setter of the type or a variable initializer).*
- **CS9117**: *Members of primary constructor parameter of a readonly type cannot be modified (except in init-only setter of the type or a variable initializer).*
- **CS9118**: *Members of primary constructor parameter of a readonly type cannot be returned by writable reference.*
- **CS9119**: *Members of primary constructor parameter of a readonly type cannot be used as a `ref` or `out` value (except in init-only setter of the type or a variable initializer).*
- **CS9120**: *Cannot return primary constructor parameter by reference.*
- **CS9121**: *Struct primary constructor parameter of type causes a cycle in the struct layout.*
- **CS9122**: *Unexpected parameter list.*
- **CS9124**: *Parameter is captured into the state of the enclosing type and its value is also used to initialize a field, property, or event.*
- **CS9136**: *Cannot use primary constructor parameter of type inside an instance member.*

Primary constructor parameters are in scope in the body of that type. The compiler can synthesize a field that stores the parameter for use in members or in field initializers. Because a primary constructor parameter may be copied to a field, the following restrictions apply:

- Primary constructors can be declared on `struct` and `class` types, but not on `interface` types.
- Primary constructor parameters can't be used in a `base()` constructor call except as part of the primary constructor.
- Primary constructor parameters of `ref struct` type can't be accessed in lambda expressions, query expressions, or local functions.
- If the type isn't a `ref struct`, `ref struct` parameters can't be accessed in instance members.

- In a `ref struct` type, primary constructor parameters with the `in`, `ref`, or `out` modifiers can't be used in any instance methods or property accessors.

Struct types have the following extra restrictions on primary constructor parameters:

- Primary constructor parameters can't be captured in lambda expressions, query expressions, or local functions.
- Primary constructor parameters can't be returned by reference (`ref` return or `readonly ref` return).

Readonly only struct types have the following extra restrictions on primary constructor parameters:

- Primary constructor parameters and their members can't be reassigned in a `readonly` struct.
- Primary constructor parameters and their members can't be `ref` returned in a `readonly` struct.
- Primary constructor parameters and their members can't be `ref` or `out` arguments to any method.

In all these cases, the restrictions on primary constructor parameters are consistent with restrictions on data fields in those types. The restrictions are because a primary constructor parameter may be transformed into a synthesized field in the type. Therefore primary constructor parameters must follow the rules that apply to that synthesized field.

A derived primary constructor calls the base primary constructor by supplying the parameters to the base constructor. You must use the parameter names from the derived constructor declaration.

The warnings provide guidance on captured or shadowed primary constructor parameters.

- **CS9107**: *Parameter is captured into the state of the enclosing type and its value is also passed to the base constructor. The value might be captured by the base class as well.* This warning indicates that your code may be allocated two copies of a primary constructor parameter. Because the parameter is passed to the base class, the base class likely uses it. Because the derived class accesses it, it may have a second copy of the same parameter. That extra storage may not be intended.
- **CS9113**: *Parameter is unread.* This warning indicates that your class never references the primary constructor, even to pass it to the base primary constructor. It likely isn't needed.

- **CS9124**: *Parameter is captured into the state of the enclosing type and its value is also used to initialize a field, property, or event.* This warning indicates that the constructor parameter of a nested type is also captured by the enclosing type. The parameter is likely stored twice.
- **CS9179**: *Primary constructor parameter is shadowed by a member from base*

Resolve errors and warnings in user-defined operator declarations

This article covers the following compiler errors:

- CS0056: *Inconsistent accessibility: return type 'type' is less accessible than operator 'operator'*
- CS0057: *Inconsistent accessibility: parameter type 'type' is less accessible than operator 'operator'*
- CS0215: *The return type of operator True or False must be bool*
- CS0216: *The operator 'operator' requires a matching operator 'missing_operator' to also be defined*
- CS0217: *In order to be applicable as a short circuit operator a user-defined logical operator ('operator') must have the same return type as the type of its 2 parameters.*
- CS0218: *The type ('type') must contain declarations of operator true and operator false*
- CS0448: *The return type for* `++` *or* `--` *operator must be the containing type or derived from the containing type*
- CS0552: *'conversion routine' : user defined conversion to/from interface*
- CS0553: *'conversion routine' : user defined conversion to/from base class*
- CS0554: *'conversion routine' : user defined conversion to/from derived class*
- CS0555: *User-defined operator cannot take an object of the enclosing type and convert to an object of the enclosing type*
- CS0556: *User-defined conversion must convert to or from the enclosing type*
- CS0557: *Duplicate user-defined conversion in type*
- CS0558: *User-defined operator must be declared static and public*
- CS0559: *The parameter type for* `++` *or* `--` *operator must be the containing type*
- CS0562: *The parameter of a unary operator must be the containing type*
- CS0563: *One of the parameters of a binary operator must be the containing type*
- CS0564: *The first operand of an overloaded shift operator must have the same type as the containing type, and the type of the second operand must be int*
- CS0567: *Interfaces cannot contain operators*
- CS0590: *User-defined operators cannot return void*
- CS0660: *Type defines* `operator ==` *or* `operator !=` *but does not override* `Object.Equals(object o)`
- CS0661: *Type defines* `operator ==` *or* `operator !=` *but does not override* `Object.GetHashCode()`
- CS0715: *Static classes cannot contain user-defined operators*
- CS1037: *Overloadable operator expected*
- CS1553: *Declaration is not valid; use 'modifier operator <dest-type> (...' instead*
- CS8930: *Explicit implementation of a user-defined operator must be static.*

- CS8931: *Explicit implementation must be declared public to implement interface member in type.*
- CS9023: *Operator cannot be made checked.*
- CS9024: *Operator cannot be made unchecked.*
- CS9025: *Operator requires a matching non-checked version to also be declared.*
- CS9308: *User-defined operator must be declared public.*
- CS9310: *The return type for this operator must be void.*
- CS9311: *Type does not implement interface member. The type cannot implement member because one of them is not an operator.*
- CS9312: *Type cannot override inherited member because one of them is not an operator.*
- CS9313: *Overloaded compound assignment operator takes one parameter.*
- CS9340: *Operator cannot be applied to operands. The closest inapplicable candidate is shown.*
- CS9341: *Operator cannot be applied to operand. The closest inapplicable candidate is shown.*
- CS9342: *Operator resolution is ambiguous between the following members.*

Operator signature requirements

- **CS0448**: *The return type for `++` or `--` operator must be the containing type or derived from the containing type.*
- **CS0559**: *The parameter type for `++` or `--` operator must be the containing type.*
- **CS0562**: *The parameter of a unary operator must be the containing type.*
- **CS0563**: *One of the parameters of a binary operator must be the containing type.*
- **CS0564**: *The first operand of an overloaded shift operator must have the same type as the containing type, and the type of the second operand must be int.*
- **CS0567**: *Interfaces can't contain operators.*
- **CS0590**: *User-defined operators can't return void.*
- **CS9310**: *The return type for this operator must be void.*
- **CS9340**: *Operator cannot be applied to operands. The closest inapplicable candidate is shown.*
- **CS9341**: *Operator cannot be applied to operand. The closest inapplicable candidate is shown.*
- **CS9342**: *Operator resolution is ambiguous between the following members.*

To declare operators with correct signatures, follow these requirements for the specific operator type. For more information, see Operator overloading.

- Return the containing type (or a derived type) from `++` and `--` operators (**CS0448**).
- Use the containing type as the parameter for `++` and `--` operators (**CS0559**).
- Use the containing type as the parameter for unary operators (**CS0562**).

- Include the containing type as at least one parameter in binary operators (**CS0563**).
- Use the containing type as the first parameter and `int` as the second parameter for shift operators (**CS0564**).
- Don't declare operators in interfaces (**CS0567**). Interfaces can't contain operator implementations.
- Return a non-void type from most operators (**CS0590**), except for specific operators that require `void` returns (**CS9310**).
- Provide operator overloads that accept the correct parameter types to avoid resolution failures (**CS9340**, **CS9341**).
- Disambiguate operator calls by using explicit casts or providing more specific overloads (**CS9342**).

> **Important**
>
> The signature requirements for static binary operators and the corresponding instance compound assignment operators are different. Make sure the signature matches the declaration you want.

The following example demonstrates signature errors:

C#

```
class C1
{
    public static int operator ++(C1 c) => 0;   // CS0448
    public static C1 operator --(C1 c) => null;   // OK
}
public class C2
{
    public static implicit operator int(C2 x) => 0;
    public static implicit operator C2(int x) => new C2();
    public static int operator ++(int aa) => 0;  // CS0559
}
public class C3
{
    public static implicit operator int(C3 x) => 0;
    public static implicit operator C3(int x) => null;
    public static C3 operator +(int aa) => 0;   // CS0562
}
public class C4
{
    public static implicit operator int(C4 x) => 0;
    public static implicit operator C4(int x) => null;
    public static int operator +(int aa, int bb) => 0;   // CS0563
}
class C5
{
```

```
    // To correct, change second operand to int, like so:
    // public static int operator << (C c1, int c2)
    public static int operator <<(C5 c1, C5 c2) => 0; // CS0564
}
interface IA
{
    int operator +(int aa, int bb);   // CS0567
}
public class C6
{
    public static void operator +(C6 A1, C6 A2) { }  // CS0590
}
```

Operator declaration requirements

- **CS0558**: *User-defined operator must be declared static and public.*
- **CS0715**: *Static classes can't contain user-defined operators.*
- **CS1037**: *Overloadable operator expected.*
- **CS1553**: *Declaration isn't valid; use 'modifier operator <dest-type> (...' instead.*
- **CS8930**: *Explicit implementation of a user-defined operator must be static.*
- **CS8931**: *Explicit implementation must be declared public to implement interface member in type.*
- **CS9308**: *User-defined operator must be declared public.*

To declare operators correctly, follow these requirements for modifiers and containing types. For more information, see Operator overloading and User-defined conversion operators.

- Declare operators with both `static` and `public` modifiers (**CS0558**, **CS9308**).
- Don't declare operators in static classes (**CS0715**). Use regular classes or structs.
- Use valid, overloadable operator symbols (**CS1037**).
- Follow the correct syntax for conversion operators: `public static implicit/explicit operator <dest-type>(<source-type> parameter)` (**CS1553**).
- Ensure explicit interface implementations of operators are `static` (**CS8930**) and `public` (**CS8931**).

The following example demonstrates declaration errors:

C#

```
public class C
{
    static implicit operator int(C aa) => 0;   // CS0558, add public
}
public static class C1
{
    public static int operator +(C1 c) => 0;  // CS0715
```

```
}
class C2
{
    public static int implicit operator (C2 f) => 6;   // CS1553
}
```

Inconsistent accessibility

- **CS0056**: *Inconsistent accessibility: return type 'type' is less accessible than operator 'operator'.*
- **CS0057**: *Inconsistent accessibility: parameter type 'type' is less accessible than operator 'operator'.*

To ensure consistent accessibility in operator declarations, make all types used in public operators publicly accessible. For more information, see Access Modifiers.

- Ensure return types have at least the same accessibility as the operator (**CS0056**).
- Ensure parameter types have at least the same accessibility as the operator (**CS0057**).

When you declare a `public` operator, all types used as parameters or return values must also be publicly accessible.

The following example demonstrates accessibility errors:

C#

```
class C { }

public class C2
{
    public static implicit operator C(C2 a) => new C();   // CS0056
}

public class C3
{
    public static implicit operator C3(C c) => new C3();   // CS0057
}
```

User-defined conversion restrictions

- **CS0552**: *User-defined conversion to/from interface.*
- **CS0553**: *User-defined conversion to/from base class.*
- **CS0554**: *User-defined conversion to/from derived class.*
- **CS0555**: *User-defined operator can't take an object of the enclosing type and convert to an object of the enclosing type.*

- **CS0556**: *User-defined conversion must convert to or from the enclosing type.*
- **CS0557**: *Duplicate user-defined conversion in type.*

To create valid user-defined conversion operators, follow these restrictions. For more information, see User-defined conversion operators.

- Don't define conversions to or from interfaces (**CS0552**). Use explicit interface implementations instead.
- Don't define conversions to or from base classes (**CS0553**). The conversion already exists through inheritance.
- Don't define conversions to or from derived classes (**CS0554**). The conversion already exists through inheritance.
- Don't define conversions from the enclosing type to itself (**CS0555**). This conversion is implicit.
- Ensure at least one type in the conversion is the enclosing type (**CS0556**). You can't define conversions between two external types.
- Don't define duplicate conversions (**CS0557**). Each conversion operator must be unique.

The following example demonstrates conversion restriction errors:

C#

```
public interface I
{
}
public class C
{
    public static implicit operator I(C aa) => default;// CS0552
}

public class B
{
}
public class D : B
{
    public static implicit operator B(D aa) => new B();// CS0553
}

public class B2
{
    // delete the conversion routine to resolve CS0554
    public static implicit operator B2(D2 d) => new B2();// CS0554
}
public class D2 : B2 { }

public class C2
{
    public static implicit operator C2(C2 aa) => new C2();   // CS0555
}
```

```
public class C3
{
    public static implicit operator int(byte aa) => 0;   // CS0556
}

public class C4
{
    public static implicit operator int(C4 aa) => 0;

    // CS0557, delete duplicate
    public static explicit operator int(C4 aa) => 0;
}
```

Boolean and short-circuit operators

- **CS0215**: *The return type of operator true or false must be bool.*
- **CS0216**: *The operator requires a matching operator to also be defined.*
- **CS0217**: *In order to be applicable as a short-circuit operator, a user-defined logical operator must have the same return type as the type of its 2 parameters.*
- **CS0218**: *The type must contain declarations of operator true and operator false.*

To define logical operators correctly, follow these pairing and signature requirements. For more information, see true and false operators, Boolean logical operators, and User-defined conditional logical operators.

- Return `bool` from `operator true` and `operator false` (**CS0215**).
- Define required paired operators (**CS0216**):
  - `operator ==` requires `operator !=`
  - `operator <` requires `operator >`
  - `operator <=` requires `operator >=`
  - `operator true` requires `operator false`
- Match the return type with the parameter types for short-circuit operators (`&` and `|`) that work with custom types (**CS0217**).
- Implement both `operator true` and `operator false` when using custom types in boolean contexts like `&&` and `||` (**CS0218**).

The following example demonstrates logical operator errors:

C#

```
class C
{
    public static int operator true(C c) => 0;   // CS0215
    public static int operator false(C c) => 0; // CS0215
```

```
}

class C2
{
    public static bool operator ==(C2 left, C2 right) => left.Equals(right);   //
CS0216

    public override bool Equals(object? o) => base.Equals(o);
    public override int GetHashCode() => base.GetHashCode();
}

public class C3
{
    public static bool operator true(C3 f) => false;
    public static bool operator false(C3 f) => true;
    public static implicit operator int(C3 x) => 0;
    public static int operator &(C3 f1, C3 f2) => new C3();  // CS0217
}

public class C4
{
    public static implicit operator int(C4 x) => 0;
    public static C4 operator &(C4 f1, C4 f2) => new C4();

    public static void Main()
    {
        C4 f = new C4();
        int i = f && f;   // CS0218, requires operators true and false
    }
}
```

Checked operators

- **CS9023**: *Operator can't be made checked*
- **CS9024**: *Operator can't be made unchecked*
- **CS9025**: *Checked operator requires a matching non-checked version to also be declared*

To use checked operators correctly, follow these requirements. For more information, see Arithmetic operators and User-defined checked operators.

- Apply `checked` or `unchecked` keywords only to supported arithmetic operators: `+`, `-`, `*`, `/`, `++`, `--`, and explicit conversions (**CS9023**, **CS9024**).
- Provide both checked and unchecked versions when declaring a checked operator (**CS9025**). The compiler needs both to handle different contexts.

Interface and inheritance requirements

- **CS9311**: *Type doesn't implement interface member. The type can't implement member because one of them isn't an operator*
- **CS9312**: *Type can't override inherited member because one of them isn't an operator*
- **CS9313**: *Overloaded compound assignment operator takes one parameter*

To implement and override operators correctly, follow these requirements. For more information, see Operator overloading and Interfaces.

- Ensure operator declarations match the signature and type of interface members (**CS9311**). An operator can't implement a non-operator member.
- Verify that inherited members being overridden are also operators (**CS9312**). An operator can't override a non-operator member.
- Declare compound assignment operators with one parameter (**CS9313**). The left operand is implicitly `this`.

Equality operators

- **CS0660**: *Type defines operator == or operator != but doesn't override Object.Equals(object o)*
- **CS0661**: *Type defines operator == or operator != but doesn't override Object.GetHashCode()*

To implement equality correctly, override the corresponding `Object` methods when defining custom equality operators. For more information, see How to define value equality for a type and Equality operators.

- Override Object.Equals when you define `operator ==` or `operator !=` (**CS0660**).
- Override Object.GetHashCode when you define `operator ==` or `operator !=` (**CS0661**).

Overriding these methods ensures consistent equality behavior across different APIs and collection types.

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

---

Last updated on 11/08/2025

Parameter and argument mismatch

Article • 12/21/2023

The compiler generates the following errors when there's no argument supplied for a formal parameter, or the argument isn't valid for that parameter:

- CS0182: *An attribute argument must be a constant expression, `typeof` expression or array creation expression of an attribute parameter type*
- CS0591: *Invalid value for argument to attribute*
- CS0599: *Invalid value for named attribute argument 'argument'*
- CS0617: *Not a valid named attribute argument. Named attribute arguments must be fields which are not readonly, static, or const, or read-write properties which are public and not static.*
- CS0633: *The argument to the attribute must be a valid identifier*
- CS0643: *Duplicate named attribute argument*
- CS0655: *Not a valid named attribute argument because it is not a valid attribute parameter type*
- CS0839: *Argument missing.*
- CS1016: *Named attribute argument expected*
- CS1739: *The best overload for does not have a parameter named*
- CS1740: *Named argument cannot be specified multiple times*
- CS1742: *An array access may not have a named argument specifier*
- CS1744: *Named argument specifies a parameter for which a positional argument has already been given*
- CS1746: *The delegate does not have a parameter named 'name'*
- CS7036: *There is no argument given that corresponds to the required parameter*
- CS7067: *Attribute constructor parameter is optional, but no default parameter value was specified.*
- CS8324: *Named argument specifications must appear after all fixed arguments have been specified in a dynamic invocation.*
- CS8905: *A function pointer cannot be called with named arguments.*
- CS8943: *null is not a valid parameter name. To get access to the receiver of an instance method, use the empty string as the parameter name.*
- CS8944: *Method is not an instance method, the receiver cannot be an interpolated string handler argument.*
- CS8945: *Not a valid parameter name.*
- CS8948: `InterpolatedStringHandlerArgumentAttribute` *arguments cannot refer to the parameter the attribute is used on.*
- CS8949: *The* `InterpolatedStringHandlerArgumentAttribute` *applied to parameter is malformed and cannot be interpreted. Construct an instance of it manually.*

- CS8950: *Parameter is an argument to the interpolated string handler conversion on parameter, but the corresponding argument is specified after the interpolated string expression. Reorder the arguments.*
- CS8951: *Parameter is not explicitly provided, but is used as an argument to the interpolated string handler conversion on parameter.*
- CS8964: *The `CallerArgumentExpressionAttribute` may only be applied to parameters with default values*
- CS8965: *The `CallerArgumentExpressionAttribute` applied to parameter will have no effect because it's self-referential.*
- CS8966: *The `CallerArgumentExpressionAttribute` will have no effect because it applies to a member that is used in contexts that do not allow optional arguments*

Missing argument

The following general errors are issued when the compiler can't match arguments to all member parameters:

- **CS0839**: *Argument missing.*
- **CS7036**: *There is no argument given that corresponds to the required parameter*

These errors are general: The compiler can't match the arguments given in a method call to the required parameters of the method. Check the following causes:

- Make sure you included all necessary arguments.
- Make sure the arguments are in the correct order.
- Make sure all arguments are the correct type.
- Make sure overload resolution rules chose the method you expected.

You might also see *CS7036* if you wrote overloaded local functions. Local functions can't be overloaded. The compiler only recognizes the first local function with that name. Check if you meant to call a different local function.

These errors often appear with other diagnostics that can help diagnose the correct cause.

Arguments for attributes

The compiler issues these errors when an argument to an attribute constructor is incorrect:

- **CS0182**: *An attribute argument must be a constant expression, `typeof` expression or array creation expression of an attribute parameter type*

- **CS0591**: *Invalid value for argument to attribute*
- **CS0599**: *Invalid value for named attribute argument 'argument'*
- **CS0617**: *Not a valid named attribute argument. Named attribute arguments must be fields which are not readonly, static, or const, or read-write properties which are public and not static.*
- **CS0633**: *The argument to the attribute must be a valid identifier*
- **CS0643**: *Duplicate named attribute argument*
- **CS0655**: *not a valid named attribute argument because it is not a valid attribute parameter type*

If you use the System.AttributeUsageAttribute on your attribute definition, make sure the allowed values aren't mutually exclusive. Check that the type and order of arguments to the attribute are correct. Make sure the text of string arguments is valid. For many attributes, the argument must be a valid C# identifier. Arguments to attribute constructors must be compile-time constants. Therefore, they're limited to types that support literal constants. In addition, the following types that allow literal constants are disallowed as attribute parameters:

- sbyte
- ushort
- uint
- ulong
- decimal

You can't specify repeated named arguments with the same parameter name. You can only set accessible properties when you initialize an attribute. You can't set private properties.

Named and optional parameters and arguments

The compiler issues the following errors for incorrect use of named and optional arguments:

- **CS1016**: *Named attribute argument expected*
- **CS1739**: *The best overload for does not have a parameter named*
- **CS1740**: *Named argument cannot be specified multiple times*
- **CS1742**: *An array access may not have a named argument specifier*
- **CS1744**: *Named argument specifies a parameter for which a positional argument has already been given*
- **CS1746**: *The delegate does not have a parameter named 'name'*

- **CS7067**: *Attribute constructor parameter is optional, but no default parameter value was specified.*
- **CS8324**: *Named argument specifications must appear after all fixed arguments have been specified in a dynamic invocation.*
- **CS8905**: *A function pointer cannot be called with named arguments.*

Check for the following causes of these errors:

- The parameter name of the named argument is incorrect.
- The chosen overload doesn't have a parameter matching the named argument.
- A parameter name is repeated on more than one argument.
- A positional (unnamed) argument appears after named arguments.
- Named arguments aren't allowed for array index parameters.

Interpolated string handler

The compiler issues the following errors when you specified an interpolated string handler incorrectly.

- **CS8943**: *null is not a valid parameter name. To get access to the receiver of an instance method, use the empty string as the parameter name.*
- **CS8944**: *Not an instance method, the receiver cannot be an interpolated string handler argument.*
- **CS8945**: *Not a valid parameter name.*
- **CS8948**: `InterpolatedStringHandlerArgumentAttribute` *arguments cannot refer to the parameter the attribute is used on.*
- **CS8949**: *The* `InterpolatedStringHandlerArgumentAttribute` *applied to parameter is malformed and cannot be interpreted. Construct an instance manually.*
- **CS8950**: *Parameter is an argument to the interpolated string handler conversion on parameter, but the corresponding argument is specified after the interpolated string expression.*
- **CS8951**: *Parameter is not explicitly provided, but is used as an argument to the interpolated string handler conversion on parameter.*

An interpolated string handler is a pattern-based construct. It's important to get the pattern correct. Consult the feature spec, or follow the tutorial on building an interpolated string handler.

Caller debugging information

The compiler issues the following error on an incorrect use of the System.Runtime.CompilerServices.CallerArgumentExpressionAttribute:

- **CS8964**: *The `CallerArgumentExpressionAttribute` may only be applied to parameters with default values*

In addition, the compiler issues the following warnings on an incorrect use of the `CallerArgumentExpressionAttribute`:

- **CS8965**: *The `CallerArgumentExpressionAttribute` applied to parameter will have no effect because it's self-referential.*
- **CS8966**: *The `CallerArgumentExpressionAttribute` will have no effect because it applies to a member that is used in contexts that do not allow optional arguments*

Any parameter annotated with the `CallerArgumentExpression` attribute must have a default value.

Resolve errors and warnings related to generic type parameters and generic type arguments

This article covers the following compiler errors:

- CS0080: *Constraints are not allowed on non-generic declarations.*
- CS0081: *Type parameter declaration must be an identifier not a type.*
- CS0224: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- CS0304: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- CS0305: *Using the generic type 'generic type' requires 'number' type arguments.*
- CS0306: *The type 'type' may not be used as a type argument.*
- CS0307: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- CS0308: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*
- CS0310: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- CS0311: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- CS0312: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- CS0313: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- CS0314: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- CS0315: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*
- CS0403: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- CS0412: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- CS0413: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*

- CS0417: *'identifier': cannot provide arguments when creating an instance of a variable type.*
- CS0694: *Type parameter 'identifier' has the same name as the containing type, or method.*
- CS0695: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- CS0698: *A generic type cannot derive from 'type' because it is an attribute class.*
- CS9338: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

Type parameter declaration and naming

The following errors relate to how type parameters are declared and named in generic types and methods:

- **CS0080**: *Constraints are not allowed on non-generic declarations.*
- **CS0081**: *Type parameter declaration must be an identifier not a type.*
- **CS0412**: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- **CS0694**: *Type parameter 'identifier' has the same name as the containing type, or method.*

To correct these errors, ensure that type parameters are declared with valid identifiers, constraint clauses are only applied to generic declarations, and type parameter names don't conflict with other identifiers in scope:

- Remove the constraint clause from non-generic declarations (**CS0080**). The `where` clause can only be used on generic types and methods that declare type parameters, because constraints define requirements that type arguments must satisfy. If you need to apply constraints, first add type parameters to your type or method declaration. For example, change `public class MyClass where MyClass : System.IDisposable` to `public class MyClass<T> where T : System.IDisposable`.
- Replace actual type names with identifiers in type parameter declarations (**CS0081**). Type parameters must be declared using identifiers (like `T`, `TKey`, or `TValue`) rather than concrete types (like `int` or `string`), because the purpose of a type parameter is to serve as a placeholder that is substituted with actual types when the generic type or method is used. For example, change `public void F<int>()` to `public void F<T>()`.
- Rename type parameters, local variables, or parameters to avoid naming conflicts (**CS0412**, **CS0694**). Type parameter names can't shadow identifiers in the same scope. They can't match the name of the containing type or method. Such conflicts create ambiguity about which identifier is being referenced. For example, if you have a method `public void F<T>()`, you can't declare a local variable `double T` inside that method, and you can't name a type parameter the same as its containing type (`class C<C>`).

For more information, see Generic Type Parameters and Generics.

Type argument count and usage

The following errors relate to providing the correct number and type of type arguments to generic types and methods:

- **CS0224**: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- **CS0305**: *Using the generic type 'generic type' requires 'number' type arguments.*
- **CS0306**: *The type 'type' may not be used as a type argument.*
- **CS0307**: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- **CS0308**: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*

To correct these errors, ensure that you provide the exact number of type arguments required by the generic declaration. Use only valid types as type arguments. Don't apply type arguments to non-generic constructs:

- Remove generic type parameters or containing generic type declarations from methods that use `__arglist` (**CS0224**). The `__arglist` keyword is incompatible with generics because the runtime mechanisms for handling variable argument lists conflict with the type substitution required for generic type parameters. This restriction also applies to the `params` keyword when used in combination with generic methods or methods within generic types.
- Supply the exact number of type arguments specified in the generic type or method declaration (**CS0305**). Each generic type parameter declared in the definition must have a corresponding type argument when the generic type is instantiated. The compiler needs to know which concrete type to substitute for each type parameter. For example, if a class is declared as `class MyList<T>`, you must provide exactly one type argument when using it, such as `MyList<int>`, not `MyList<int, string>`.
- Use only valid types as type arguments (**CS0306**). Pointer types, such as `int*` or `char*`, can't be used as type arguments because generic types require managed types that the garbage collector can track, and pointer types are unmanaged. If you need to work with pointers in a generic context, consider using `IntPtr` or restructuring your code to avoid mixing generics with unsafe code.
- Remove type argument syntax from non-generic constructs (**CS0307**, **CS0308**). Type arguments enclosed in angle brackets (like `<int>`) can only be applied to generic types and methods that declare type parameters. You must either remove the type arguments entirely or ensure you imported the namespace that contains the generic version of the type. For example, `IEnumerator<T>` requires the `using System.Collections.Generic;` directive, whereas `IEnumerator` is in `System.Collections`.

For more information, see Generic Type Parameters and Generics.

Constructor constraints

The following errors relate to the `new()` constraint on generic type parameters:

- **CS0304**: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- **CS0310**: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- **CS0417**: *'identifier': cannot provide arguments when creating an instance of a variable type.*

To correct these errors, add the `new()` constraint to type parameters that need to be instantiated, ensure type arguments have public parameterless constructors, and avoid passing arguments when constructing instances of type parameters:

- Add the `new()` constraint to the type parameter declaration (**CS0304**). When you use the `new` operator to create an instance of a type parameter within a generic type or method, the compiler must be able to guarantee that any type argument supplied at runtime has a parameterless constructor available. The `new()` constraint provides this guarantee at compile time, allowing the compiler to generate the appropriate instantiation code. For example, if you have `class C<T>` with a member `T t = new T();`, you must change the declaration to `class C<T> where T : new()`.
- Ensure that type arguments used with `new()` constrained type parameters have public parameterless constructors (**CS0310**). When a generic type or method declares a `new()` constraint on a type parameter, any concrete type used as a type argument must be non-abstract and must provide a public parameterless constructor. If a type only has non-public constructors (such as `private` or `protected`) or only has constructors with parameters, it can't satisfy the `new()` constraint. To fix this error, either add a public parameterless constructor to the type, or use a different type argument that already has one.
- Remove constructor arguments when instantiating type parameters (**CS0417**). The `new()` constraint only guarantees the existence of a parameterless constructor, so attempting to pass arguments to `new T(arguments)` isn't allowed because the compiler can't verify that a constructor with those specific parameter types exists on the type that are substituted for `T`. If you need to construct instances with specific arguments, consider using factory methods, abstract factory patterns, or specific base class/interface constraints that define the construction behavior you need.

For more information, see Constraints on type parameters and the new() constraint.

Constraint satisfaction and conversions

The following errors relate to type arguments not satisfying the constraints of generic type parameters:

- **CS0311**: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- **CS0312**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- **CS0313**: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- **CS0314**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- **CS0315**: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*

To correct these errors, use type arguments that satisfy all constraints through appropriate conversions, ensure derived classes repeat base class constraints, and understand that nullable value types have special constraint requirements:

- Change the type argument to one that has an implicit reference conversion to the constraint type (**CS0311**). When a type parameter has a constraint like `where T : BaseType`, any type argument must be convertible to `BaseType` through an implicit reference conversion or identity conversion. The type argument must either be `BaseType` itself, derive from `BaseType`, or implement `BaseType` if it's an interface. Implicit numeric conversions (such as from `short` to `int`) don't satisfy generic type parameter constraints because these conversions are value conversions, not reference conversions.
- Repeat the base class's type parameter constraints in any derived class declaration (**CS0314**). When a derived generic class inherits from a base generic class that has constraints on its type parameters, the derived class must declare the same constraints on its corresponding type parameters. Repetition is required because the compiler needs to verify that type arguments supplied to the derived class satisfies the requirements of the base class. For example, if you have `public class A<T> where T : SomeClass`, then any class deriving from it must be declared as `public class B<T> : A<T> where T : SomeClass`.
- Use non-nullable value types or change the constraint type (**CS0312**, **CS0313**). Nullable value types (such as `int?`) are distinct from their underlying value types and don't satisfy the same constraints. There's no implicit conversion between `int?` and `int`, and nullable

value types can't satisfy interface constraints because the nullable wrapper itself doesn't implement the interface, even though the underlying value type does. To fix these errors, either use the non-nullable form of the value type as the type argument, or adjust your constraint to accept `object` or a nullable reference type if appropriate.

- Ensure type arguments satisfy reference type or class constraints (**CS0315**). When a type parameter is constrained to a class type (such as `where T : SomeClass`), you can't use a value type (struct) as the type argument because there's no boxing conversion that satisfies the constraint relationship. The constraint requires a reference type that has an inheritance or implementation relationship with the constraint type. To resolve this error, either change the struct to a class if semantically appropriate, or remove the class constraint if the generic type can work with value types.

For more information, see Constraints on type parameters and Implicit conversions.

Generic type usage restrictions

The following errors relate to restrictions on how generic types can be used:

- **CS0403**: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- **CS0413**: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*
- **CS0695**: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- **CS0698**: *A generic type cannot derive from 'type' because it is an attribute class.*
- **CS9338**: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

To correct these errors, use `default` instead of `null` for unconstrained type parameters, add class constraints when using the `as` operator, avoid interface unification conflicts, don't create generic attribute classes, and ensure type arguments match the visibility of their containing members:

- Replace `null` assignments with `default(T)` or add a class constraint (**CS0403**). When you assign `null` to an unconstrained type parameter, the compiler can't guarantee that the type argument is a reference type that accepts `null` values, because it might be a value type like `int` or `struct`, which can't be `null`. To resolve this error, either use `default(T)`, which provides the appropriate default value for any type (null for reference types, zero or empty for value types), or add a `class` constraint to the type parameter if you specifically need reference type semantics and want to allow `null` assignments.
- Add a `class` or specific type constraint when using the `as` operator (**CS0413**). The `as` operator performs a safe type cast that returns `null` if the conversion fails, but this

behavior is incompatible with value types because value types can't be `null`. When you use `as` with an unconstrained type parameter, the compiler can't guarantee the type argument isn't a value type, so it rejects the code. To fix this error, add a `class` constraint or a specific reference type constraint (like `where T : SomeClass`) to ensure the type parameter is always a reference type that can properly handle the `null` result of a failed cast.

- Avoid implementing the same generic interface multiple times with type parameters that could unify (**CS0695**). When a class implements a generic interface multiple times with different type parameters (such as `class G<T1, T2> : I<T1>, I<T2>`), there's a risk that someone could instantiate it with the same type for both parameters (`G<int, int>`), which would create a conflict because the class would effectively be implementing `I<int>` twice. To resolve this error, either implement the interface only once, restructure your type parameters to prevent unification, or use separate non-generic classes for different specializations.
- Remove generic type parameters from attribute classes (**CS0698**).

  > **Note**
  >
  > This error is no longer produced in current versions of C#, as generic attributes are now supported.

- Ensure type arguments used in public or protected signatures are at least as accessible as the member using them (**CS9338**). A public or protected generic member must use type arguments that are publicly accessible. Otherwise external code couldn't properly reference or use the member's signature. For example, if you have `public class Container<T>` where `T` is an internal type, external assemblies can see the `Container` but can't properly work with it because they can't see `T`. To fix this error, either make the type argument public, or reduce the accessibility of the member using it to match the type argument's accessibility.

For more information, see Constraints on type parameters, default value expressions, and Attributes.

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

Last updated on 11/14/2025

Resolve errors and warnings in async methods using the await operator

This article covers the following compiler errors:

- CS1983: *Since this is an async method, the return expression must be of type '`Task<T>`' rather than '`T`'.*
- CS1985: *Cannot await in a catch clause.*
- CS1986: *'`await`' requires that the type have a suitable '`GetAwaiter`' method.*
- CS1989: *Async lambda expressions cannot be converted to expression trees.*
- CS1991: *'Type' cannot implement 'event' because it is a Windows Runtime event and 'event' is a regular .NET event.*
- CS1992: *The '`await`' operator can only be used when contained within a method or lambda expression marked with the 'async' modifier.*
- CS1994: *The '`async`' modifier can only be used in methods that have a body.*
- CS1995: *The '`await`' operator may only be used in a query expression within the first collection expression of the initial '`from`' clause or within the collection expression of a '`join`' clause.*
- CS1996: *Cannot await in the body of a lock statement.*
- CS1997: *Since function is an async method that returns a value, a return keyword must not be followed by an object expression.*
- CS1998: *This async method lacks '`await`' operators and will run synchronously. Consider using the '`await`' operator to await non-blocking API calls, or '`await Task.Run(...)`' to do CPU-bound work on a background thread.*
- CS4008: *Cannot await '`void`'.*
- CS4009: *A void or int returning entry point cannot be async.*
- 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.*
- CS4032: *The '`await`' operator can only be used within an async method. Consider marking this method with the 'async' modifier and changing its return type to '`Task<T>`'.*
- CS4033: *The '`await`' operator can only be used within an async method. Consider marking this method with the '`async`' modifier and changing its return type to '`Task`'.*
- CS8892: *Method will not be used as an entry point because a synchronous entry point was found.*
- CS9123: *The '`&`' operator should not be used on parameters or local variables in async methods.*
- CS9330: *'`MethodImplAttribute.Async`' cannot be manually applied to methods. Mark the method 'async'.*

Await expression requirements

- **CS1985**: *Cannot await in a catch clause.*
- **CS1986**: '`await`' *requires that the type have a suitable* '`GetAwaiter`' *method.*
- **CS1992**: *The* '`await`' *operator can only be used when contained within a method or lambda expression marked with the* '`async`' *modifier.*
- **CS1995**: *The* '`await`' *operator may only be used in a query expression within the first collection expression of the initial* '`from`' *clause or within the collection expression of a* '`join`' *clause.*
- **CS1996**: *Cannot await in the body of a lock statement.*
- **CS4008**: *Cannot await* '`void`'.
- **CS4032**: *The* '`await`' *operator can only be used within an async method. Consider marking this method with the* '`async`' *modifier and changing its return type to* '`Task<T>`'.
- **CS4033**: *The* '`await`' *operator can only be used within an async method. Consider marking this method with the* '`async`' *modifier and changing its return type to* '`Task`'.

To use the `await` operator correctly, follow these rules. For more information, see Asynchronous programming with async and await.

- Don't use `await` in catch clauses (**CS1985**). While you can use `await` in try blocks and finally blocks (in C# 6 and later), catch blocks present special challenges with exception handling and control flow.
- Don't use `await` inside lock statement blocks (**CS1996**). The compiler doesn't support this to avoid emitting code prone to deadlocks.
- Use `await` only in specific locations within query expressions (**CS1995**): within the first collection expression of the initial `from` clause, or within the collection expression of a `join` clause.
- Mark methods or lambda expressions with the `async` modifier before using `await` (**CS1992**, **CS4032**, **CS4033**).
- Ensure awaited types have an accessible `GetAwaiter` method that returns an awaiter type (**CS1986**).
- Don't apply `await` to expressions of type `void` (**CS4008**).
- Change the return type to `Task` for methods that don't return a value, or `Task<T>` for methods that return a value.

Async method signature requirements

- **CS1983**: *Since this is an async method, the return expression must be of type* '`Task<T>`' *rather than* '`T`'.

- **CS1994**: *The '`async`' modifier can only be used in methods that have a body.*
- **CS4009**: *A void or int returning entry point cannot be async.*
- **CS8892**: *Method will not be used as an entry point because a synchronous entry point was found.*
- **CS9330**: *'`MethodImplAttribute.Async`' cannot be manually applied to methods. Mark the method '`async`'.*

To declare async methods correctly, follow these signature requirements. For more information, see Async main return values.

- Return one of the valid types: `void`, Task, `Task<T>`, a task-like type, IAsyncEnumerable<T>, or IAsyncEnumerator<T> (**CS1983**).
- Use the `async` modifier only on methods with a body (**CS1994**). Remove the `async` modifier on abstract methods in interfaces or classes.
- Update to C# 7.1 or higher to use `async` on the `Main` entry point, or avoid using `async` on entry points in earlier versions (**CS4009**).
- Remove synchronous entry points if you have both sync and async entry points (**CS8892**).
- Use the `async` keyword instead of manually applying `MethodImplAttribute.Async` (**CS9330**).

Async practices

- **CS1989**: *Async lambda expressions cannot be converted to expression trees.*
- **CS1991**: *'Type' cannot implement 'event' because it is a Windows Runtime event and 'event' is a regular .NET event.*
- **CS1997**: *Since function is an async method that returns a value, a return keyword must not be followed by an object expression.*
- **CS1998**: *This async method lacks '`await`' operators and will run synchronously. Consider using the '`await`' operator to await non-blocking API calls, or '`await Task.Run(...)`' to do CPU-bound work on a background thread.*
- **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.*
- **CS9123**: *The '`&`' operator should not be used on parameters or local variables in async methods.*

To write async code correctly and avoid common pitfalls, follow these best practices. For more information, see Asynchronous programming with async and await.

- Always await calls to async methods that return Task or Task<TResult> (**CS4014**). Unawaited calls can lead to lost exceptions and unexpected behavior.

- Don't return a value from async methods that return `Task` (non-generic); use `Task<T>` instead (**CS1997**).
- Include at least one `await` operator in async methods, or remove the `async` modifier (**CS1998**).
- Remove the `return` statement if the method should return `Task` (**CS1997**, **CS1998**).
- Change the method's return type to `Task<T>` to return a value (**CS1997**, **CS1998**).
- Remove the `async` modifier and return the task directly if you don't need the async state machine (**CS1997**, **CS1998**).
- Don't use async methods in expression trees (**CS1989**). Expression trees represent code as data and don't support the complex state machine transformations required by async methods.
- Don't mark add or remove accessors in an interface or WinRT event as async (**CS1991**). This is a platform-specific restriction for Windows Runtime interoperability.
- Avoid using the address-of operator (`&`) on expressions inside async methods (**CS9123**). The target may be relocated in memory during suspension, making the pointer invalid.

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

---

Last updated on 11/11/2025

Resolve errors and warnings related to members that implement an interface

This article covers the following compiler errors:

- CS0071: *An explicit interface implementation of an event must use event accessor syntax.*
- CS0106: *The modifier is not valid for this item.*
- CS0277: *Member does not implement interface member because it is not public.*
- CS0425: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- CS0460: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*
- CS0470: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- CS0473: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- CS0531: *Interface members cannot have a definition.*
- CS0535: *Member does not implement interface member.*
- CS0538: *Member in explicit interface declaration is not an interface.*
- CS0539: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- CS0540: *Containing type does not implement interface member.*
- CS0541: *Explicit interface declaration can only be declared in a class, record, struct or interface.*
- CS0550: *Member adds an accessor not found in interface member.*
- CS0551: *Explicit interface implementation is missing an accessor.*
- CS0630: *Member cannot implement interface member because it has an __arglist parameter.*
- CS0686: *Accessor cannot implement interface member. Use an explicit interface implementation.*
- CS0736: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- CS0737: *Member does not implement interface member. It cannot implement an interface member because it is not public.*
- CS0738: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- CS8705: *Interface member does not have a most specific implementation. Neither member is most specific.*

- CS8854: *Member does not implement interface member.*
- CS9333: *Parameter type must match implemented member.*
- CS9334: *Return type must match implemented member.*

Interface declaration and syntax

The following errors relate to proper syntax and structure when declaring explicit interface implementations:

- **CS0071**: *An explicit interface implementation of an event must use event accessor syntax.*
- **CS0106**: *The modifier is not valid for this item.*
- **CS0531**: *Interface members cannot have a definition.*
- **CS0538**: *Member in explicit interface declaration is not an interface.*
- **CS0541**: *Explicit interface declaration can only be declared in a class, record, struct or interface.*

You can correct these errors using the following techniques:

- You must manually provide `add` and `remove` event accessors when explicitly implementing an interface event (**CS0071**). The compiler doesn't automatically generate these accessors for explicit interface implementations, so you must define them explicitly to specify how the event is stored and managed.
- Remove the `public` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations are implicitly public when accessed through the interface type, making the `public` keyword redundant and not allowed in this context.
- Remove the `abstract` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations provide the actual implementation and can't be marked as abstract because they can't be overridden in derived classes.
- Remove the method body from interface member declarations, or move the implementation to a class or struct that implements the interface (**CS0531**). Before C# 8.0, interface members can't contain implementations; starting with C# 8.0, you can provide default interface methods using specific syntax.
- Verify that the type specified in the explicit interface declaration is an actual interface type (**CS0538**). Only interface types can be used in explicit interface implementation syntax; attempting to use a class or other non-interface type violates the explicit implementation rules.
- Move explicit interface declarations into a class or struct that declares the interface in its base list (**CS0541**). Explicit interface implementations must appear within the body of a class or struct type and can't be declared at the namespace level or in other contexts.

For more information, see Interfaces, Explicit Interface Implementation, and How to implement interface events.

Return types and signatures

The following errors occur when the implementing method's signature doesn't match the interface member declaration:

- **CS0738**: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- **CS8854**: *Member does not implement interface member.*
- **CS9333**: *Parameter type must match implemented member.*
- **CS9334**: *Return type must match implemented member.*

You can correct these errors using the following techniques:

- Change the return type of the implementing method to exactly match the return type declared in the interface member (**CS0738**, **CS9334**). The signature of the implementation must match the interface declaration precisely because the method signature is part of the contract that determines which interface member is being implemented.
- Ensure that parameter types in the implementing method exactly match the parameter types declared in the interface member (**CS9333**). Each parameter must have the identical type in the same position as specified in the interface declaration, as parameter types are fundamental components of the method signature that the compiler uses to match implementations to interface members.
- Add an `init` accessor to the implementing property when the interface property declares an `init` setter (**CS8854**). The `init` keyword allows property initialization during object construction while preventing modification afterward, and the implementing property must provide this same initialization-only behavior to satisfy the interface contract.

For more information, see Interfaces, Properties, and Init-only setters.

Missing or incomplete implementations

The following errors occur when a class fails to fully implement an interface or implements members that don't match the interface contract:

- **CS0535**: *Member does not implement interface member.*
- **CS0550**: *Member adds an accessor not found in interface member.*
- **CS0551**: *Explicit interface implementation is missing an accessor.*

You can correct these errors using the following techniques:

- Provide an implementation for every member declared in the interface, or declare the type as `abstract` (**CS0535**). Each member must be implemented to satisfy the interface requirements.
- Remove any accessors from the implementing property that aren't declared in the interface property (**CS0550**). The implementing property can only include the accessors explicitly declared in the interface definition, ensuring that the implementation doesn't add functionality beyond what the interface contract specifies.
- Add all required accessors to the explicit interface implementation to match the interface declaration (**CS0551**). Each accessor declared in the interface must have a corresponding accessor in the implementation with matching signatures, as the implementation must fulfill the complete accessor contract defined by the interface.

For more information, see Interfaces and Properties.

Member matching and resolution

The following errors occur when attempting to implement interface members that don't exist in the interface or when the containing type doesn't declare the interface:

- **CS0539**: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- **CS0540**: *Containing type does not implement interface member.*

You can correct these errors using the following techniques:

- Verify that the member name and signature in the explicit interface implementation exactly match a member declared in the interface, or remove the incorrect implementation (**CS0539**). The member you're attempting to implement must actually exist in the interface definition with matching name, return type, and parameter types, as explicit interface implementation requires precise correspondence with the interface contract.
- Add the interface to the class's or struct's base list, or remove the explicit interface implementation (**CS0540**). A type can only explicitly implement members of interfaces that it declares in its inheritance list, so the implementing type must establish the interface relationship before it can provide explicit implementations.

For more information, see Interfaces and Explicit Interface Implementation.

Generic type constraints

The following errors occur when implementing generic interface methods with type parameter constraints:

- **CS0425**: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- **CS0460**: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*

You can correct these errors using the following techniques:

- Ensure the `where` clause in the implementing method is identical to the interface method declaration, or matches the semantic meaning of the constraints (**CS0425**). The type parameter constraints in the implementation must match those defined in the interface or base method.
- Remove explicit constraint declarations from override and explicit interface implementation methods (**CS0460**). The override method inherits its constraints automatically from the base or interface method, so redeclaring them is redundant and not permitted except for specific cases allowed in C# 8 and later.
- Apply the `default` constraint to resolve ambiguities with nullable reference types in override and explicit interface implementations when using C# 9 or later (**CS0460**). This exception to the constraint inheritance rule allows you to explicitly specify the default constraint to disambiguate nullable annotation contexts.
- Explicitly specify `where T : class` or `where T : struct` constraints on override and explicit interface implementation methods when using C# 8 or later to enable nullable reference type annotations (**CS0460**). These specific constraints are permitted to support nullable reference type analysis on type parameters that are constrained to reference or value types.

For more information, see Constraints on type parameters, Interfaces, and Nullable reference types.

Method visibility and modifiers

The following errors occur when implementing interface methods with incorrect accessibility or modifiers:

- **CS0736**: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- **CS0737**: *Member does not implement interface member. It cannot implement an interface member because it is not public.*

You can correct these errors using the following techniques:

- Remove the `static` modifier from the method declaration that implements the interface member (**CS0736**). Before C# 10, interface members are instance members, not static members.
- Add the `public` access modifier to the method that implements the interface member (**CS0737**). All interface members are implicitly `public` because interfaces define a contract for public behavior, so the implementing method must also have public accessibility to be accessible through the interface reference.

For more information, see Interfaces and Access Modifiers.

Accessor implementation and conflicts

The following errors occur when implementing interface properties or events with accessor methods that have visibility issues or naming conflicts:

- **CS0277**: *Member does not implement interface member because it is not public.*
- **CS0470**: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- **CS0686**: *Accessor cannot implement interface member. Use an explicit interface implementation.*

You can correct these errors using the following techniques:

- Remove any access modifiers from property accessors that restrict visibility to less visible than `public`, or add the `public` modifier if it's missing (**CS0277**). All interface members are implicitly `public`, so the implementing accessor must also have public accessibility to satisfy the interface contract and be accessible through the interface type.
- Replace methods with accessor-like names (such as `get_PropertyName`) with proper property syntax using explicit interface implementation (**CS0470**). The compiler generates accessor methods internally for properties, and attempting to manually create methods with these reserved names conflicts with the property implementation mechanism.
- Use explicit interface implementation syntax to resolve naming conflicts when the interface contains method names that match the auto-generated accessor methods (**CS0686**). The compiler automatically generates methods like `get_Property` and `set_Property` for properties, and `add_Event` and `remove_Event` for events, so if an interface declares methods with these exact names, explicit implementation is required to disambiguate between the interface method and the compiler-generated accessor.

For more information, see Interfaces, Properties, and Events.

Ambiguous and conflicting implementations

The following errors occur when the compiler can't determine which interface implementation to use:

- **CS0473**: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- **CS8705**: *Interface member 'member' does not have a most specific implementation. Neither is most specific.*

You can correct these errors using the following techniques:

- Eliminate the explicit interface implementation and instead use a single implicit public implementation for both interface methods (**CS0473**). When a generic method acquires the same signature as a non-generic method (such as when implementing `ITest<int>` where both `TestMethod(int)` and `TestMethod(T)` become identical), the common language infrastructure metadata system can't unambiguously determine which interface member binds to which implementation slot, so using implicit implementation allows the single method to satisfy both interface requirements.
- Provide an explicit implementation in the implementing class or struct that resolves the ambiguity between multiple default implementations (**CS8705**). This error typically occurs with diamond inheritance patterns where a class implements multiple interfaces that each provide default implementations for the same member. The compiler needs you to explicitly specify which implementation to use, or provide your own implementation.
- Restructure the interface hierarchy to avoid diamond inheritance conflicts where multiple interfaces provide default implementations for the same member (**CS8705**). By redesigning the interface relationships or consolidating the default implementations into a single interface, you can eliminate the ambiguity that prevents the compiler from determining the most specific implementation.

For more information, see Interfaces and Default Interface Methods.

Special implementation restrictions

The following error occurs when using special parameter types that aren't compatible with interface implementation:

- **CS0630**: *Member cannot implement interface member because it has an __arglist parameter.*

You can correct this error using the following techniques:

- Remove the `__arglist` parameter from the implementing method (**CS0630**). The `__arglist` keyword allows methods to accept variable numbers of arguments in an unmanaged way, but this feature is incompatible with interface implementation because interface contracts require predictable, type-safe signatures that can be verified at compile time.
- Replace the `__arglist` parameter with a `params` array parameter for variable-length argument lists (**CS0630**). Unlike `__arglist`, the `params` keyword provides a type-safe mechanism for accepting variable numbers of arguments that is fully compatible with interface implementation and maintains the compile-time type safety that interfaces require.

For more information, see Interfaces and params keyword.

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

---

Last updated on 11/14/2025

Errors and warnings associated with reference parameters, variables, and returns

Article • 11/08/2024

The following errors can be generated when you're working with reference variables:

- CS0192: *A `readonly` field cannot be used as a `ref` or `out` value (except in a constructor)*
- CS0199: *A `static readonly` field cannot be used as a `ref` or `out` value (except in a static constructor)*
- CS0206: *A non ref-returning property or indexer may not be used as an `out` or `ref` value*
- CS0631: *`ref` and `out` are not valid in this context*
- CS0767: *Cannot inherit interface with the specified type parameters because it causes method to contain overloads which differ only on `ref` and `out`*
- CS1510: *A `ref` or `out` value must be an assignable variable*
- CS1605: *Cannot use variable as a `ref` or `out` value because it is read-only*
- CS1623: *Iterators cannot have `ref`, `in` or `out` parameters*
- CS1649: *Members of a `readonly` field cannot be used as a `ref` or `out` value (except in a constructor)*
- CS1651: *Fields of a static readonly field cannot be used as a `ref` or `out` value (except in a static constructor)*
- CS1655: *Cannot use fields of type as a `ref` or `out` value*
- CS1657: *Cannot use variable as a `ref` or `out` value*
- CS1741: *A `ref` or `out` parameter cannot have a default value*
- CS1939: *Cannot pass the range variable as an `out` or `ref` parameter*
- CS1988: *Async methods cannot have `ref`, `in` or `out` parameters*
- CS7084: *A Windows Runtime event may not be passed as an `out` or `ref` parameter.*
- CS8166: *Cannot return a parameter by reference because it is not a `ref` parameter*
- CS8167: *Cannot return by reference a member of parameter because it is not a `ref` or `out` parameter*
- CS8168: *Cannot return local by reference because it is not a ref local*
- CS8169: *Cannot return a member of local variable by reference because it is not a ref local*
- CS8196: *Reference to an implicitly-typed out variable is not permitted in the same argument list.*

- CS8325: *'`await`' cannot be used in an expression containing a `ref` conditional operator*
- CS8326: *Both conditional operator values must be ref values or neither may be a ref value*
- CS8327: *The expression must be of correct type to match the alternative ref value*
- CS8329: *Cannot use variable as a `ref` or `out` value because it is a readonly variable*
- CS8330: *Members of variable cannot be used as a `ref` or `out` value because it is a readonly variable*
- CS8331: *Cannot assign to variable or use it as the right hand side of a `ref` assignment because it is a readonly variable*
- CS8332: *Cannot assign to a member of variable or use it as the right hand side of a `ref` assignment because it is a readonly variable*
- CS8337: *The first parameter of a '`ref`' extension method must be a value type or a generic type constrained to struct.*
- CS8338: *The first '`in`' or '`ref readonly`' parameter of the extension method must be a concrete (non-generic) value type.*
- CS8351: *Branches of a `ref` conditional operator cannot refer to variables with incompatible declaration scopes*
- CS8373: *The left-hand side of a `ref` assignment must be a ref variable.*
- CS8374: *Cannot ref-assign source has a narrower escape scope than destination.*
- CS8388: *An `out` variable cannot be declared as a ref local*
- CS8977: *Cannot use '`ref`', '`in`', or '`out`' in the signature of a method attributed with 'UnmanagedCallersOnly'.*
- CS9072: *A deconstruction variable cannot be declared as a ref local*
- CS9077: *Cannot return a parameter by reference through a `ref` parameter; it can only be returned in a return statement*
- CS9078: *Cannot return by reference a member of parameter through a `ref` parameter; it can only be returned in a return statement*
- CS9079: *Cannot ref-assign because source can only escape the current method through a return statement.*
- CS9096: *Cannot ref-assign because source has a wider value escape scope than destination allowing assignment through source of values with narrower escapes scopes than destination.*
- CS9101: *UnscopedRefAttribute can only be applied to struct or virtual interface instance methods and properties, and cannot be applied to constructors or init-only members.*
- CS9102: *UnscopedRefAttribute cannot be applied to an interface implementation because implemented member doesn't have this attribute.*

- CS9104: *A `using` statement resource of type cannot be used in async methods or async lambda expressions.*
- CS9190: *`readonly` modifier must be specified after `ref`.*
- CS9199: *A `ref readonly` parameter cannot have the Out attribute.*

The following warnings are generated when reference variables are used incorrectly:

- CS9085: *This ref-assigns variable but destination has a narrower escape scope than source.*
- CS9086: *The branches of the `ref` conditional operator refer to variables with incompatible declaration scopes*
- CS9087: *This returns a parameter by reference but it is not a `ref` parameter*
- CS9089: *This returns by reference a member of parameter that is not a `ref` or `out` parameter*
- CS9091: *This returns local by reference but it is not a ref local*
- CS9092: *This returns a member of local by reference but it is not a ref local*
- CS9093: *This ref-assigns but source can only escape the current method through a return statement.*
- CS9094: *This returns a parameter by reference through a `ref` parameter; but it can only safely be returned in a return statement*
- CS9095: *This returns by reference a member of parameter through a `ref` parameter; but it can only safely be returned in a return statement*
- CS9097: *This ref-assigns but source has a wider value escape scope than destination allowing assignment through destination of values with narrower escapes scopes than source.*
- CS9191: *The `ref` modifier for argument corresponding to `in` parameter is equivalent to `in`. Consider using `in` instead.*
- CS9192: *Argument should be passed with `ref` or `in` keyword.*
- CS9193: *Argument should be a variable because it is passed to a `ref readonly` parameter*
- CS9195: *Argument should be passed with the `in` keyword*
- CS9196: *Reference kind modifier of parameter doesn't match the corresponding parameter in overridden or implemented member.*
- CS9197: *Reference kind modifier of parameter doesn't match the corresponding parameter in hidden member.*
- CS9198: *Reference kind modifier of parameter doesn't match the corresponding parameter in target.*
- CS9200: *A default value is specified for `ref readonly` parameter, but `ref readonly` should be used only for references. Consider declaring the parameter as `in`.*
- CS9201: *Ref field should be ref-assigned before use.*

- CS9265: *Field is never ref-assigned to, and will always have its default value (null reference)*

These errors and warnings follow these themes:

- ***Incorrect syntax***: The syntax of your declaration or usage is invalid.
- ***Language constructs where ref variables aren't valid***: Some C# idioms don't allow variables. Usually this is because ref safety analysis can't be performed reliably.
- ***Value expression used where a reference variable is needed***: The expression used as a reference variable must be a variable, not a value expression.
- ***Writable reference variables referring to readonly variables***: A reference to a readonly variable can't be passed by writable reference.
- ***violations of ref safety***: A reference variable can't refer to a variable that has a narrower context. That would mean the reference variable could refer to invalid memory.

This article uses the term *reference variable* as a general term for a parameter declared with one of the `in`, `ref readonly`, `ref`, or `out` modifiers, or a `ref` local variable, a `ref` field in a `ref struct`, or a `ref` return. A reference variable refers to another variable, called the *referent*.

Incorrect syntax

These errors indicate that you're using incorrect syntax regarding reference variables:

- **CS8373**: *The left-hand side of a `ref` assignment must be a ref variable.*
- **CS8388**: *An `out` variable cannot be declared as a ref local.*
- **CS9190**: *`readonly` modifier must be specified after `ref`.*

You can correct the error with one of these changes:

- The left operand of an `= ref` operator must be a reference variable. For more information on the correct syntax, see reference variables.
- The parameter modifier `ref readonly` must be in that order. `readonly ref` is not a legal parameter modifier. Switch the order of the words.
- A local variable can't be declared as `out`. To declare a local reference variable, use `ref`.

Reference variable restrictions

The following errors indicate that a reference variable can't be used where you have one:

- **CS0631**: *`ref` and `out` are not valid in this context*
- **CS0767**: *Cannot inherit interface with the specified type parameters because it causes method to contain overloads which differ only on `ref` and `out`*
- **CS1623**: *Iterators cannot have `ref`, `in` or `out` parameters*
- **CS1741**: *A `ref` or `out` parameter cannot have a default value*
- **CS1939**: *Cannot pass the range variable as an `out` or `ref` parameter*
- **CS1988**: *Async methods cannot have `ref`, `in` or `out` parameters*
- **CS7084**: *A Windows Runtime event may not be passed as an `out` or `ref` parameter.*
- **CS8196**: *Reference to an implicitly-typed `out` variable is not permitted in the same argument list.*
- **CS8325**: *'await' cannot be used in an expression containing a `ref` conditional operator*
- **CS8326**: *Both conditional operator values must be ref values or neither may be a ref value*
- **CS8327**: *The expression must be of correct type to match the alternative ref value*
- **CS8337**: *The first parameter of a '`ref`' extension method must be a value type or a generic type constrained to struct.*
- **CS8338**: *The first '`in`' or '`ref readonly`' parameter of the extension method must be a concrete (non-generic) value type.*
- **CS8977**: *Cannot use '`ref`', '`in`', or '`out`' in the signature of a method attributed with 'UnmanagedCallersOnly'.*
- **CS9072**: *A deconstruction variable cannot be declared as a ref local*
- **CS9104**: *A `using` statement resource of type cannot be used in async methods or async lambda expressions.*
- **CS9199**: *A `ref readonly` parameter cannot have the Out attribute.*

The following warnings indicate that a reference variable shouldn't be used, and might be unsafe:

- **CS9196**: *Reference kind modifier of parameter doesn't match the corresponding parameter in overridden or implemented member.*
- **CS9197**: *Reference kind modifier of parameter doesn't match the corresponding parameter in hidden member.*
- **CS9198**: *Reference kind modifier of parameter doesn't match the corresponding parameter in target.*
- **CS9200**: *A default value is specified for `ref readonly` parameter, but `ref readonly` should be used only for references. Consider declaring the parameter as `in`.*

- **CS9201**: *Ref field should be ref-assigned before use.*
- **CS9265**: *Field is never ref-assigned to, and will always have its default value (null reference)*

To fix the error, remove the reference variable where it isn't allowed:

- Remove `in`, `ref`, and `out` parameters from indexers, iterators, and async methods.
- Remove ref conditional expressions (`? :`) that include an await.
- Remove the `ref` modifier from the first parameter of a extension method where that type isn't a value type or a generic type constrained as a value type.
- Either both or neither [conditional operator expressions] must be `ref` variables. Either remove `ref` from one expression, or add it to the other. If it's a `ref` conditional expression, both expressions must be the same type.
- `ref` and `out` parameters can't have default values. Either remove the `ref` or `out` modifier, or remove the default value.
- An implicitly typed `out` variable declaration can't also appear elsewhere in the same argument list.
- You can't put reference variables in a `using` statement in `async` methods lambda expressions.
- The range variable in a LINQ query expression can't be passed by reference.
- You can't deconstruct an object into reference variables. Replace the reference variables with value variables.
- You can't implement multiple interfaces where method overloads differ only on `ref` and `out`. For example, one interface declares `void M(ref int i)` and another interface declares `void M(out int i)`. A class can't implement both interfaces because the methods aren't distinguishable. You can only implement one of those interfaces.
- Methods attributed with System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute can't use reference parameters.
- A Windows runtime event can't be passed as a reference variable.
- A `ref readonly` parameter can't have the System.Runtime.InteropServices.OutAttribute applied to it in remoting API.
- Initialize a `ref` field in the constructor or as a field initializer.

`unscoped ref` restrictions

The `unscoped` qualifier on `ref` parameters isn't allowed in some locations:

- **CS9101**: *UnscopedRefAttribute can only be applied to struct instance or virtual interface methods and properties, and cannot be applied to constructors or or init-only members.*
- **CS9102**: *UnscopedRefAttribute cannot be applied to an interface implementation because implemented member doesn't have this attribute..*

You must remove the `unscoped` modifier on the parameter declaration that caused the error.

Reference variables require a referent

You must supply a variable as an argument to a reference parameter, reference return, or ref local assignment:

- **CS0206**: *A non ref-returning property or indexer may not be used as an `out` or `ref` value*
- **CS1510**: *A `ref` or `out` value must be an assignable variable*

Warnings:

- **CS9191**: *The `ref` modifier for argument corresponding to `in` parameter is equivalent to `in`. Consider using `in` instead.*
- **CS9192**: *Argument should be passed with `ref` or `in` keyword.*
- **CS9193**: *Argument should be a variable because it is passed to a `ref readonly` parameter*
- **CS9195**: *Argument should be passed with the `in` keyword*

The compiler emits these errors when you use an expression that calculates a value where a variable must be used. You must store the result of that expression in a variable to use it. For example, properties and indexers return values, not variables. You must store the result in a variable and pass a reference to that variable.

Writable reference variables require a writable referent

A writable reference variable requires that the referent also is writable. The following errors indicate that the variable isn't writable:

- **CS0192**: *A `readonly` field cannot be used as a `ref` or `out` value (except in a constructor)*

- **CS0199**: *A `static readonly` field cannot be used as a `ref` or `out` value (except in a static constructor)*
- **CS1605**: *Cannot use variable as a `ref` or `out` value because it is read-only*
- **CS1649**: *Members of a `readonly` field cannot be used as a `ref` or `out` value (except in a constructor)*
- **CS1651**: *Fields of a `static readonly` field cannot be used as a `ref` or `out` value (except in a static constructor)*
- **CS1655**: *Cannot use fields of type as a `ref` or `out` value*
- **CS1657**: *Cannot use variable as a `ref` or `out` value*
- **CS8329**: *Cannot use variable as a `ref` or `out` value because it is a readonly variable*
- **CS8330**: *Members of variable cannot be used as a `ref` or `out` value because it is a readonly variable*
- **CS8331**: *Cannot assign to variable or use it as the right hand side of a `ref` assignment because it is a readonly variable*
- **CS8332**: *Cannot assign to a member of variable or use it as the right hand side of a `ref` assignment because it is a readonly variable*

Examples of variables that aren't writable include:

- readonly fields, both instance and static fields.
- Members of `readonly` fields.
- The `this` variable.
- The foreach iteration variable
- A using variable, or a fixed variable.

You must copy the value and pass a reference to the copy.

Ref safety violations

The compiler tracks the safe context of referents and reference variables. The compiler issues errors, or warnings in unsafe code, when a reference variable refers to a referent variable that's no longer valid. The referent must have a safe context that is at least as wide as the ref safe context of The reference variable. Violating these safety checks means the reference variable accesses random memory instead of the referent variable.

- **CS8166**: *Cannot return a parameter by reference because it is not a `ref` parameter*
- **CS8167**: *Cannot return by reference a member of parameter because it is not a `ref` or `out` parameter*
- **CS8168**: *Cannot return local by reference because it is not a ref local*
- **CS8169**: *Cannot return a member of local variable by reference because it is not a ref local*

- **CS8345**: *Field or auto-implemented property cannot be of type unless it is an instance member of a `ref struct`.*
- **CS8351**: *Branches of a `ref` conditional operator cannot refer to variables with incompatible declaration scopes*
- **CS8374**: *Cannot ref-assign source has a narrower escape scope than destination.*
- **CS9077**: *Cannot return a parameter by reference through a `ref` parameter; it can only be returned in a return statement*
- **CS9078**: *Cannot return by reference a member of parameter through a `ref` parameter; it can only be returned in a return statement*
- **CS9079**: *Cannot ref-assign source to destination because source can only escape the current method through a return statement.*
- **CS9096**: *Cannot ref-assign source to destination because source has a wider value escape scope than destination allowing assignment through destination of values with narrower escapes scopes than source.*

Warnings:

- **CS9085**: *This ref-assigns source to destination but source has a narrower escape scope than destination.*
- **CS9086**: *The branches of the ref conditional operator refer to variables with incompatible declaration scopes*
- **CS9087**: *This returns a parameter by reference but it is not a `ref` parameter*
- **CS9089**: *This returns by reference a member of parameter that is not a `ref` or `out` parameter*
- **CS9091**: *This returns local by reference but it is not a ref local*
- **CS9092**: *This returns a member of local by reference but it is not a ref local*
- **CS9093**: *This ref-assigns source to destination but source can only escape the current method through a return statement.*
- **CS9094**: *This returns a parameter by reference through a `ref` parameter; but it can only safely be returned in a return statement*
- **CS9095**: *This returns by reference a member of parameter through a `ref` parameter; but it can only safely be returned in a return statement*
- **CS9097**: *This ref-assigns source to destination but source has a wider value escape scope than destination allowing assignment through destination of values with narrower escapes scopes than source.*

The compiler uses static analysis to determine if the referent is valid at all points where the reference variable can be used. You need to refactor code so that the referent remains valid at all locations where the reference variable might refer to it. For details on the rules for ref safety, see the C# standard on ref safe contexts.

Errors and warnings associated with `ref struct` types

Article • 11/08/2024

- CS8343: *`ref structs` cannot implement interfaces*
- CS8344: *`foreach` statement cannot operate on enumerators in async or iterator methods because type is a `ref struct` or a type parameter that allows `ref struct`.*
- CS8345: *Field or auto-implemented property cannot be of type unless it is an instance member of a `ref struct`.*
- CS9048: *The `scoped` modifier can be used for refs and `ref struct` values only.*
- CS9050: *A `ref` field cannot refer to a `ref struct`.*
- CS9059: *A ref field can only be declared in a ref struct.*
- CS9241: *'ref struct' is already specified.*
- CS9242: *The 'allows' constraint clause must be the last constraint specified.*
- CS9243: *Cannot allow ref structs for a type parameter known from other constraints to be a class.*
- CS9244: *The type may not be a `ref struct` or a type parameter allowing ref structs in order to use it as parameter in the generic type or method.*
- CS9245: *Type cannot implement interface member for `ref struct` type.*
- CS9246: *A non-virtual instance interface member cannot be accessed on a type parameter that allows ref struct.*
- CS9247: *foreach statement cannot operate on enumerators of type because it is a type parameter that allows ref struct and it is not known at compile time to implement `IDisposable`.*
- CS9267: *Element type of an iterator may not be a ref struct or a type parameter allowing ref structs*

ref safety violations

- **CS8345**: *Field or auto-implemented property cannot be of type unless it is an instance member of a `ref struct`.*
- **CS9048**: *The `scoped` modifier can be used for refs and `ref struct` values only.*
- **CS9050**: *A `ref` field cannot refer to a `ref struct`.*
- **CS9059**: *A `ref` field can only be declared in a `ref struct`.*

A ref struct type can include `ref` fields. Other types aren't allowed `ref` fields. The compiler enforces restrictions on the declarations and use of `ref struct` types to enforce ref safety rules on instances of any `ref struct` type:

- Only `ref struct` types can contain automatically implemented `ref` properties.
- Only `ref struct` types or `ref` variables can have the `scoped` modifier.
- A `ref` field can be declared only in a `ref struct` type.
- A `ref` field can't refer to a `ref struct` type/

Violating any of these rules produces one of the listed errors. If you intended to use that language feature, convert the type to a `ref struct`. Otherwise, remove the disallowed construct.

ref struct interface implementations

- **CS8343**: *`ref structs` cannot implement interfaces*
- **CS8344**: *`foreach` statement cannot operate on enumerators in async or iterator methods because type is a `ref struct` or a type parameter that allows `ref struct`.*
- **CS9241**: *'ref struct' is already specified.*
- **CS9242**: *The 'allows' constraint clause must be the last constraint specified.*
- **CS9243**: *Cannot allow ref structs for a type parameter known from other constraints to be a class.*
- **CS9244**: *The type may not be a `ref struct` or a type parameter allowing ref structs in order to use it as parameter in the generic type or method.*
- **CS9245**: *Type cannot implement interface member for `ref struct` type.*
- **CS9246**: *A non-virtual instance interface member cannot be accessed on a type parameter that allows ref struct.*
- **CS9247**: *foreach statement cannot operate on enumerators of type because it is a type parameter that allows ref struct and it is not known at compile time to implement `IDisposable`.*
- **CS9267**: *Element type of an iterator may not be a ref struct or a type parameter allowing ref structs*

Prior to C# 13, ref struct types can't implement interfaces; the compiler generates *CS8343*. Beginning with C# 13, `ref struct` types can implement interfaces, subject to the following rules:

- A `ref struct` can't be converted to an instance of an interface it implements. This restriction includes the implicit conversion when you use a `ref struct` type as an argument when the parameter is an interface type. The conversion results in a boxing conversion, which violates ref safety.
- A `ref struct` that implements an interface *must* implement all interface members. The `ref struct` must implement members where the interface includes a default implementation.

Beginning with C# 13, a `ref struct` can be used as a type argument for a generic type parameter, if and only if the generic type parameter has the allows ref struct anti-constraint. When you use the `allows ref struct` anti-constraint you must follow these rules:

- A `ref struct` is used as a type argument, the type parameter *must* have the `allows ref struct` anti-constraint.- The `allows ref struct` anti-constraint must be last in the `where` clause for that parameter
- Uses of instances the type parameter must obey ref safety rules.
- A `ref struct` or a type argument that can be a `ref struct` type can't be used as the element type for an iterator method.

Errors and warnings related to the `yield return` statement and iterator methods

Article • 12/18/2024

There are numerous *errors* related to the `yield return` statement and iterator methods:

- CS1622: *Cannot return a value from an iterator. Use the yield return statement to return a value, or yield break to end the iteration.*
- CS1624: *The body of 'accessor' cannot be an iterator block because 'type' is not an iterator interface type*
- CS1625: *Cannot yield in the body of a finally clause*
- CS1626: *Cannot yield a value in the body of a try block with a catch clause*
- CS1627: *Expression expected after yield return*
- CS1629: *Unsafe code may not appear in iterators*
- CS1631: *Cannot yield a value in the body of a catch clause*
- CS1637: *Iterators cannot have unsafe parameters or yield types*
- CS4013: *Instance of type cannot be used inside a nested function, query expression, iterator block or async method*
- CS8154: *The body cannot be an iterator block because it returns by reference*
- CS8176: *Iterators cannot have by-reference locals*
- CS9238: *Cannot use 'yield return' in an 'unsafe' block*
- CS9239: *The `&` operator cannot be used on parameters or local variables in iterator methods.*

Structure of an iterator method

An iterator method must conform to several rules in C#. The compiler issues the following errors when your iterator method violates one or more of those rules:

- **CS1622**: *Cannot return a value from an iterator. Use the yield return statement to return a value, or yield break to end the iteration.*
- **CS1624**: *The body of 'accessor' cannot be an iterator block because 'type' is not an iterator interface type*
- **CS1627**: *Expression expected after yield return*
- **CS1637**: *Iterators cannot have unsafe parameters or yield types*
- **CS8154**: *The body cannot be an iterator block because it returns by reference*

Your iterator method must follow the following rules:

- An iterator method (using `yield return` and optionally `yield break`) can't also use a `return` statement to return a sequence.
- An iterator method must declare an *iterator interface type* as the return type. The *iterator interface types* are: IEnumerable, IEnumerable<T>, IEnumerator, IEnumerator<T>.
- A `yield return` statement must include an expression to return as part of a sequence. `yield return;` isn't valid.
- An iterator method can't use unsafe types as parameters, such as pointers.
- An iterator method can't `yield return` unsafe type, such as pointers.
- An iterator method can't `yield return` by `ref`. You must return by value.

Restrictions on iterator methods

The body of an iterator method must conform to restrictions on the `yield return` statement and its context. The compiler issues the following errors when your iterator violates one of these restrictions:

- **CS1625**: *Cannot yield in the body of a finally clause*
- **CS1626**: *Cannot yield a value in the body of a try block with a catch clause*
- **CS1631**: *Cannot yield a value in the body of a catch clause*
- **CS1629**: *Unsafe code may not appear in iterators*
- **CS9238**: *Cannot use 'yield return' in an 'unsafe' block*
- **CS9239**: *The `&` operator cannot be used on parameters or local variables in iterator methods.*

These errors indicate that your code violates safety rules because an iterator returns an element and resumes to generate the next element:

- You can't `yield return` from a `catch` or `finally` clause.
- You can't `yield return` from a `try` block with a catch clause.
- You can't `yield return` from an `unsafe` block. The context for an iterator creates a nested `safe` block within the enclosing `unsafe` block.
- You can't use the `&` operator to take the address of a variable in an iterator method.

Before C# 13, iterators can't contain `unsafe` code (CS1629). Beginning with C# 13, this restriction is relaxed. All `yield return` statements must be in a safe context, but an iterator method can contain `unsafe` code.

ref safety in iterator methods

Iterator methods have special ref safety restrictions. These rules are relaxed in C# 13:

- **CS4013**: *Instance of type cannot be used inside a nested function, query expression, iterator block or async method*
- **CS8176**: *Iterators cannot have by-reference locals*

Before C# 13, iterators couldn't declare `ref` local variables. They could not declare any variables of a `ref struct` type.

Beginning with C# 13, `ref struct` types can be used in iterator methods, if they aren't accessed across `yield return` statement. Beginning with C# 13, iterator methods can declare `ref` local variables.

Resolve errors and warnings in extension member declarations

This article covers the following compiler errors:

- CS1100: *Method has a parameter modifier '`this`' which is not on the first parameter*
- CS1101: *The parameter modifier '`ref`' cannot be used with '`this`'.*
- CS1102: *The parameter modifier '`out`' cannot be used with '`this`'.*
- CS1103: *The first parameter of an extension method cannot be of a pointer type.*
- CS1105: *Extension methods must be static.*
- CS1106: *Extension methods must be defined in a non generic static class.*
- CS1109: *Extension Methods must be defined on top level static classes, 'name' is a nested class.*
- CS1110: *Cannot define a new extension because the compiler required type ExtensionAttribute cannot be found. Are you missing a reference to System.Core.dll?*
- CS1112: *Do not use 'ExtensionAttribute'. Use the '`this`' keyword instead.*
- CS1113: *Extension method defined on a value type cannot be used to create delegates.*
- CS1743: *Cannot specify a default value for the 'this' parameter.*
- CS9281: *Extension declarations may not have a name.*
- CS9282: *Extension declarations can include only methods or properties.*
- CS9283: *Extensions must be declared in a top-level, non-generic, static class.*
- CS9284: *The receiver parameter of an extension cannot have a default value.*
- CS9285: *An extension container can have only one receiver parameter.*
- CS9287: *A receiver parameter cannot have the same name as an extension container type parameter.*
- CS9288: *A parameter, local variable, or local function cannot have the same name as an extension container type parameter.*
- CS9289: *Member type parameter has the same name as an extension container type parameter.*
- CS9290: *A parameter, local variable, or local function cannot have the same name as an extension parameter.*
- CS9291: '`value`': *an automatically-generated parameter name conflicts with an extension parameter name.*
- CS9292: *A type parameter has the same name as an extension parameter.*
- CS9293: *Cannot use an extension parameter in this context.*
- CS9294: '`value`': *an automatically-generated parameter name conflicts with an extension type parameter name.*
- CS9295: *The extended type must reference all the type parameters declared by the extension, but a type parameter is not referenced.*

- CS9300: *The '`ref`' receiver parameter of an extension block must be a value type or a generic type constrained to struct.*
- CS9301: *The '`in`' or '`ref readonly`' receiver parameter of extension must be a concrete (non-generic) value type.*
- CS9302: *new protected member declared in an extension block.*
- CS9303: *Cannot declare instance members in an extension block with an unnamed receiver parameter.*
- CS9304: *Cannot declare init-only accessors in an extension block.*
- CS9305: *Cannot use modifiers on the unnamed receiver parameter of extension block.*
- CS9306: *Types and aliases cannot be named 'extension'.*
- CS9309: *An extension member syntax is disallowed in nested position within an extension member syntax.*
- CS9316: *Extension members are not allowed as an argument to '`nameof`'.*
- CS9317: *The parameter of a unary operator must be the extended type.*
- CS9318: *The parameter type for ++ or -- operator must be the extended type.*
- CS9319: *One of the parameters of a binary operator must be the extended type.*
- CS9320: *The first operand of an overloaded shift operator must have the same type as the extended type.*
- CS9321: *An extension block extending a static class cannot contain user-defined operators.*
- CS9322: *Cannot declare instance operator for a struct unless containing extension block receiver parameter is a '`ref`' parameter.*
- CS9323: *Cannot declare instance extension operator for a type that is not known to be a struct and is not known to be a class.*
- CS9326: *'`name`': extension member names cannot be the same as their extended type.*
- CS9329: *This extension block collides with another extension block. They result in conflicting content-based type names in metadata.*
- CS9339: *The extension resolution is ambiguous between the following members.*

Common errors on extension declarations

- **CS1102**: *The parameter modifier '`out`' cannot be used with '`this`'.*
- **CS1103**: *The first parameter of an extension method cannot be of a pointer type.*
- **CS1106**: *Extension methods must be defined in a non generic static class.*
- **CS1109**: *Extension Methods must be defined on top level static classes, 'name' is a nested class.*
- **CS1113**: *Extension method defined on a value type cannot be used to create delegates.*
- **CS1743**: *Cannot specify a default value for the 'this' parameter.*
- **CS9283**: *Extensions must be declared in a top-level, non-generic, static class.*
- **CS9284**: *The receiver parameter of an extension cannot have a default value.*
- **CS9285**: *An extension container can have only one receiver parameter.*

The compiler emits these errors when you violate rules that apply to all extension member declarations, regardless of the syntax chosen. For more information, see Extension methods.

To declare extension members correctly, follow these requirements:

- Declare the containing type as a non-generic `static` class or struct (**CS1106**, **CS9283**).
- Declare the containing type at the top level, not nested within another type (**CS1109**, **CS9283**).
- Don't convert extension methods on value types to delegates (**CS1113**). Create a regular method instead.
- Don't use the `out` parameter modifier on the receiver parameter (**CS1102**).
- Don't provide default values for the receiver parameter (**CS1743**, **CS9284**).
- Don't extend pointer types (**CS1103**). The parameter you apply the `this` modifier to can't be a pointer type.
- Declare only one receiver parameter per extension container (**CS9285**).

Errors related to extension block declarations

- **CS9281**: *Extension declarations may not have a name.*
- **CS9282**: *Extension declarations can include only methods or properties.*
- **CS9287**: *A receiver parameter cannot have the same name as an extension container type parameter.*
- **CS9288**: *A parameter, local variable, or local function cannot have the same name as an extension container type parameter.*
- **CS9289**: *Member type parameter has the same name as an extension container type parameter.*
- **CS9290**: *A parameter, local variable, or local function cannot have the same name as an extension parameter.*
- **CS9291**: '`value`': *an automatically-generated parameter name conflicts with an extension parameter name.*
- **CS9292**: *A type parameter has the same name as an extension parameter.*
- **CS9293**: *Cannot use an extension parameter in this context.*
- **CS9294**: '`value`': *an automatically-generated parameter name conflicts with an extension type parameter name.*
- **CS9295**: *The extended type must reference all the type parameters declared by the extension, but a type parameter is not referenced.*
- **CS9300**: *The '`ref`' receiver parameter of an extension block must be a value type or a generic type constrained to struct.*
- **CS9301**: *The '`in`' or '`ref readonly`' receiver parameter of extension must be a concrete (non-generic) value type.*

- **CS9302**: *new protected member declared in an extension block.*
- **CS9303**: *Cannot declare instance members in an extension block with an unnamed receiver parameter.*
- **CS9304**: *Cannot declare init-only accessors in an extension block.*
- **CS9305**: *Cannot use modifiers on the unnamed receiver parameter of extension block.*
- **CS9306**: *Types and aliases cannot be named 'extension'.*
- **CS9309**: *An extension member syntax is disallowed in nested position within an extension member syntax.*
- **CS9316**: *Extension members are not allowed as an argument to '`nameof`'.*
- **CS9317**: *The parameter of a unary operator must be the extended type.*
- **CS9318**: *The parameter type for ++ or -- operator must be the extended type.*
- **CS9319**: *One of the parameters of a binary operator must be the extended type.*
- **CS9320**: *The first operand of an overloaded shift operator must have the same type as the extended type.*
- **CS9321**: *An extension block extending a static class cannot contain user-defined operators.*
- **CS9322**: *Cannot declare instance operator for a struct unless containing extension block receiver parameter is a '`ref`' parameter.*
- **CS9323**: *Cannot declare instance extension operator for a type that is not known to be a struct and is not known to be a class.*
- **CS9326**: *'`name`': extension member names cannot be the same as their extended type.*
- **CS9329**: *This extension block collides with another extension block. They result in conflicting content-based type names in metadata.*
- **CS9339**: *The extension resolution is ambiguous between the following members.*

These errors are specific to extension blocks, a C# 14 feature. Extension blocks are declared using the extension contextual keyword in a static class. For more information, see Extension methods.

To declare extension blocks correctly, follow these requirements:

- Don't include a name token in the extension declaration (**CS9281**). The extension declares the receiver only.
- Don't provide default values for the receiver parameter (**CS9284**, covered in common errors).
- Don't use the `extension` keyword for types or aliases (**CS9306**). It's a contextual keyword for extension blocks only.

To declare extension members in extension blocks correctly, follow these requirements in addition to the common rules:

- Include only methods or properties as extension members (**CS9282**). Other member types aren't supported.

- Provide a parameter name for the receiver to contain instance extension members (**CS9303**).
- Ensure the receiver parameter name is unique within the extension block and doesn't conflict with type parameters (**CS9287**, **CS9288**, **CS9289**, **CS9290**, **CS9291**, **CS9292**, **CS9294**).
- Reference all type parameters declared on the extension in the extended type (**CS9295**). Additional type parameters can be added on individual members.
- Don't nest extension blocks within other extension blocks (**CS9309**).
- Use the `ref` modifier on the receiver parameter only with value types or generic types constrained to struct (**CS9300**).
- Use the `in` or `ref readonly` modifier on the receiver parameter only with concrete (non-generic) value types (**CS9301**).
- Don't use modifiers on unnamed receiver parameters (**CS9305**).
- Don't declare `protected` members in extension blocks (**CS9302**). Extension members must be accessible where the extension is in scope.
- Don't declare `init`-only accessors in extension blocks (**CS9304**). Use regular property setters instead.
- Don't use extension members as arguments to the `nameof` operator (**CS9316**).
- Choose member names that differ from the extended type name (**CS9326**).
- Ensure extension blocks have unique content-based type names in metadata (**CS9329**). Consolidate or differentiate extension blocks to avoid conflicts.
- Resolve ambiguous extension member calls by providing more specific type information or using qualified names (**CS9339**).

Extension block operator requirements

Extension blocks support user-defined operators with specific requirements:

- Unary operators must have the extended type as their parameter (**CS9317**).
- Increment (`++`) and decrement (`--`) operators must have the extended type as their parameter (**CS9318**).
- Binary operators must have at least one parameter that is the extended type (**CS9319**).
- Shift operators must have the extended type as their first operand (**CS9320**).
- Don't declare user-defined operators in extension blocks that extend static classes (**CS9321**).
- When extending a struct with instance operators, use the `ref` modifier on the receiver parameter (**CS9322**).
- Don't declare instance operators for types that aren't constrained to be either a struct or a class (**CS9323**).

Errors related to `this` parameter extension methods

- **CS1100**: *Method has a parameter modifier '`this`' which is not on the first parameter*
- **CS1101**: *The parameter modifier '`ref`' cannot be used with '`this`'.*
- **CS1105**: *Extension methods must be static.*
- **CS1110**: *Cannot define a new extension because the compiler required type ExtensionAttribute cannot be found. Are you missing a reference to System.Core.dll?*
- **CS1112**: *Do not use 'ExtensionAttribute'. Use the '`this`' keyword instead.*

These errors are specific to extension methods where you declare the receiver by adding the `this` modifier to the first parameter. For more information, see Extension methods.

To declare `this` parameter extension methods correctly, follow these requirements in addition to the common rules:

- Add the `static` modifier to the method (**CS1105**).
- Apply the `this` parameter modifier only to the first parameter (**CS1100**).
- Don't combine the `ref` modifier with the `this` modifier (**CS1101**). To use `ref`, convert to an extension block.
- Add a reference to `System.Core.dll` in .NET Framework apps (**CS1110**).
- Use the `this` modifier on the first parameter instead of directly applying the ExtensionAttribute attribute (**CS1112**).

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

---

Last updated on 11/08/2025

Errors and warnings related to `partial` type and `partial` member declarations

05/31/2025

There are numerous *errors* related to `partial` type and `partial` member declarations:

- CS0260: *Missing partial modifier on declaration of type; another partial declaration of this type exists*
- CS0261: *Partial declarations of type must be all classes, all structs, or all interfaces*
- CS0262: *Partial declarations of type have conflicting accessibility modifiers*
- CS0263: *Partial declarations of type must not specify different base classes*
- CS0264: *Partial declarations of type must have the same type parameter names in the same order*
- CS0265: *Partial declarations of type have inconsistent constraints for type parameter 'type parameter'*
- CS0267: *The '`partial`' modifier can only appear immediately before '`class`', '`record`', '`struct`', '`interface`', or a method or property return type.*
- CS0282: *There is no defined ordering between fields in multiple declarations of partial class or struct 'type'. To specify an ordering, all instance fields must be in the same declaration.*
- CS0501: *'member function' must declare a body because it is not marked `abstract`, `extern`, or `partial`*
- CS0750: *A partial member cannot have the '`abstract`' modifier.*
- CS0751: *A partial member must be declared in a partial `class` or partial `struct`*
- CS0754: *A partial member may not explicitly implement an interface method.*
- CS0755: *Both partial method declarations must be extension methods or neither may be an extension method.*
- CS0756: *A partial method may not have multiple defining declarations.*
- CS0757: *A partial method may not have multiple implementing declarations.*
- CS0759: *No defining declaration found for implementing declaration of partial method.*
- CS0761: *Partial method declarations of `method<T>` have inconsistent type parameter constraints.*
- CS0762: *Cannot create delegate from method because it is a partial method without an implementing declaration*
- CS0763: *Both partial method declarations must be `static` or neither may be `static`.*
- CS0764: *Both partial method declarations must be `unsafe` or neither may be `unsafe`*
- CS1067: *Partial declarations must have the same type parameter names and variance modifiers in the same order.*
- CS8142: *Both partial member declarations must use the same tuple element names.*
- CS8663: *Both partial member declarations must be readonly or neither may be readonly*

- CS8796: *Partial method must have accessibility modifiers because it has a non-void return type.*
- CS8795: *Partial member must have an implementation part because it has accessibility modifiers.*
- CS8797: *Partial method must have accessibility modifiers because it has '`out`' parameters.*
- CS8798: *Partial method must have accessibility modifiers because it has a '`virtual`', '`override`', '`sealed`', '`new`', or '`extern`' modifier.*
- CS8799: *Both partial member declarations must have identical accessibility modifiers.*
- CS8800: *Both partial member declarations must have identical combinations of `virtual`, `override`, `sealed`, and `new` modifiers.*
- CS8817: *Both partial method declarations must have the same return type.*
- CS8818: *Partial member declarations must have matching `ref` return values.*
- CS8863: *Only a single partial type declaration may have a parameter list*
- CS8988: *The `scoped` modifier of parameter doesn't match partial definition.*
- CS9248: *Partial property must have an implementation part.*
- CS9249: *Partial property must have a definition part.*
- CS9250: *A partial property may not have multiple defining declarations, and cannot be an auto-property.*
- CS9251: *A partial property may not have multiple implementing declarations*
- CS9252: *Property accessor must be implemented because it is declared on the definition part*
- CS9253: *Property accessor does not implement any accessor declared on the definition part*
- CS9254: *Property accessor must match the definition part*
- CS9255: *Both partial property declarations must have the same type.*
- CS9256: *Partial property declarations have signature differences.*
- CS9257: *Both partial property declarations must be required or neither may be required*
- CS9258: *In this language version, 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.*
- CS9263: *A partial property cannot have an initializer on both the definition and implementation.*
- CS9275: *Partial member must have an implementation part.*
- CS9276: *Partial member must have a definition part.*
- CS9277: *Partial member may not have multiple defining declarations.*
- CS9278: *Partial member may not have multiple implementing declarations.*
- CS9279: *Partial event cannot have initializer.*
- CS9280: *Only the implementing declaration of a partial constructor can have an initializer.*

The following warnings can be generated for field backed properties:

- CS9264: *Non-nullable property must contain a non-null value when exiting constructor. Consider adding the 'required' modifier, or declaring the property as nullable, or adding '`[field: MaybeNull, AllowNull]`' attributes.**
- CS9266: *One accessor of property should use '`field`' because the other accessor is using it.*
- CS9273: *In this language version, '`field`' is a keyword within a property accessor. Rename the variable or use the identifier '`@field`' instead.*

The following sections explain the cause and fixes for these errors and warnings.

Partial types

- CS0260: *Missing partial modifier on declaration of type; another partial declaration of this type exists*
- CS0261: *Partial declarations of type must be all classes, all structs, or all interfaces*
- CS0262: *Partial declarations of type have conflicting accessibility modifiers*
- CS0263: *Partial declarations of type must not specify different base classes*
- CS0264: *Partial declarations of type must have the same type parameter names in the same order*
- CS0265: *Partial declarations of type have inconsistent constraints for type parameter 'type parameter'*
- CS0267: *The '`partial`' modifier can only appear immediately before '`class`', '`record`', '`struct`', '`interface`', or a method or property return type.*
- CS8863: *Only a single partial type declaration may have a parameter list*

Your partial type declaration can cause the compiler to emit the following warning:

- CS0282: *There is no defined ordering between fields in multiple declarations of partial `class` or `struct` 'type'. To specify an ordering, all instance fields must be in the same declaration.*

For any partial type, the `partial` keyword must immediately precede `class`, `record`, `struct`, or `interface`. The compiler emits an error if it appears in any other order. In addition:

- All declarations of a partial type must match in terms of the type (`class`, `struct`, `record class`, `record struct`, `readonly struct`, or `readonly record struct`).
- All declarations must include the `partial` modifier.
- The declarations for a generic partial type must include the same type parameters in the same order.

Some parts of the declaration aren't required to be repeated on all declarations for a type. However, if these elements are repeated on multiple `partial` declarations, they must match:

- Any access modifiers, such as `public`.
- Any base class or implemented interfaces.
- Any constraints on type parameters.

A primary constructor can be declared on at most one declaration for a partial type.

The compiler warns you if you have multiple fields declared in multiple files for a `partial struct` type. If the layout order is important, you must declare all fields in the same file. If order doesn't matter, you can use the System.Runtime.InteropServices.StructLayoutAttribute with the LayoutKind.Auto value.

Partial members

- **CS0750**: *A partial member cannot have the '`abstract`' modifier.*
- **CS0751**: *A partial member must be declared in a partial `class` or partial `struct`*
- **CS0754**: *A partial member may not explicitly implement an interface method.*
- **CS0763**: *Both partial method declarations must be `static` or neither may be `static`.*
- **CS0764**: *Both partial method declarations must be `unsafe` or neither may be `unsafe`*
- **CS8142**: *Both partial member declarations must use the same tuple element names.*
- **CS8663**: *Both partial member declarations must be readonly or neither may be readonly*
- **CS8799**: *Both partial member declarations must have identical accessibility modifiers.*
- **CS8800**: *Both partial member declarations must have identical combinations of `virtual`, `override`, `sealed`, and `new` modifiers.*
- **CS8818**: *Partial member declarations must have matching `ref` return values.*
- **CS8988**: *The `scoped` modifier of parameter doesn't match partial definition.*
- **CS9275**: *Partial member must have an implementation part.*
- **CS9276**: *Partial member must have a definition part.*
- **CS9277**: *Partial member may not have multiple defining declarations.*
- **CS9278**: *Partial member may not have multiple implementing declarations.*

Partial members must have two declarations. The declaration without an implementation is the *defining declaration*. The declaration with the implementation is the *implementing declaration*. Both declarations are required. Partial members are allowed only in a `partial` type. Partial members can't be `abstract`. Partial members can't explicitly implement an interface. Both declarations of a partial member must have identical signatures. For example, either both or neither declarations can include the `static` or `unsafe` modifiers.

Partial methods

- **CS0501**: *'member function' must declare a body because it is not marked `abstract`, `extern`, or `partial`*
- **CS0755**: *Both partial method declarations must be extension methods or neither may be an extension method.*
- **CS0756**: *A partial method may not have multiple defining declarations.*
- **CS0757**: *A partial method may not have multiple implementing declarations.*
- **CS0759**: *No defining declaration found for implementing declaration of partial method.*
- **CS0761**: *Partial method declarations of `method<T>` have inconsistent type parameter constraints.*
- **CS0762**: *Cannot create delegate from method because it is a partial method without an implementing declaration*
- **CS1067**: *Partial declarations must have the same type parameter names and variance modifiers in the same order.*
- **CS8796**: *Partial method must have accessibility modifiers because it has a non-void return type.*
- **CS8795**: *Partial member must have an implementation part because it has accessibility modifiers.*
- **CS8797**: *Partial method must have accessibility modifiers because it has '`out`' parameters.*
- **CS8798**: *Partial method must have accessibility modifiers because it has a '`virtual`', '`override`', '`sealed`', '`new`', or '`extern`' modifier.*
- **CS8817**: *Both partial method declarations must have the same return type.*

Certain `partial` method declarations don't require an *implementing declaration*. That is, if the member returns `void`, doesn't declare any access modifiers (including the default `private` modifier), and doesn't include any of the `virtual`, `override`, `sealed`, or `new` modifiers. Otherwise, any partial method must include both the declaring and implementing declarations.

When a partial method includes an implementing declaration, both declarations must be identical. Exactly one implementing declaration can be defined.

Partial properties

The following errors indicate mistakes in your partial property or indexer declarations:

- **CS9248**: *Partial property must have an implementation part.*
- **CS9249**: *Partial property must have a definition part.*
- **CS9250**: *A partial property may not have multiple defining declarations, and cannot be an auto-property.*
- **CS9251**: *A partial property may not have multiple implementing declarations*
- **CS9252**: *Property accessor must be implemented because it is declared on the definition part*

- **CS9253**: *Property accessor does not implement any accessor declared on the definition part*
- **CS9254**: *Property accessor must match the definition part*
- **CS9255**: *Both partial property declarations must have the same type.*
- **CS9257**: *Both partial property declarations must be required or neither may be required*

The following warning indicates a signature difference in the declaring and implementing declarations in a partial property:

- **CS9256**: *Partial property declarations have signature differences.*

A partial property or indexer must have both a *declaring declaration* and an *implementing declaration*. The signatures for both declarations must match. Because the *declaring declaration* uses the same syntax as an automatically implemented property, the *implementing declaration* can't be an automatically implemented property. The accessors must have at least one accessor body. Beginning in C# 13, you can use the field keyword to declare one accessor using a concise syntax:

```
C#

public partial int ImplementingDeclaration { get => field; set; }
```

Partial events and constructors

- **CS9279**: *Partial event cannot have initializer.*
- **CS9280**: *Only the implementing declaration of a partial constructor can have an initializer.*

You've declared an initializer on the defining declaration of a partial constructor or on a partial event declaration. You must remove it.

field backed properties

- **CS9258**: *In this language version, 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.*
- **CS9263**: *A partial property cannot have an initializer on both the definition and implementation.*
- **CS9264**: *Non-nullable property must contain a non-null value when exiting constructor. Consider adding the 'required' modifier, or declaring the property as nullable, or adding '`[field: MaybeNull, AllowNull]`' attributes.**
- **CS9266**: *One accessor of property should use '`field`' because the other accessor is using it.*

- **CS9273**: *In this language version, '`field`' is a keyword within a property accessor. Rename the variable or use the identifier '`@field`' instead.*

> **ⓘ Important**
>
> The `field` keyword is a preview feature in C# 13. You must be using .NET 9 and set your `<LangVersion>` element to `preview` in your project file in order to use the `field` contextual keyword.
>
> 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.

Beginning with C# 13, the preview feature, `field` backed properties allows you to access the compiler synthesized backing field for a property. **CS9258** or **CS9273** indicate that you have a variable named `field`, which can be hidden by the contextual keyword `field`.

CS9263 indicates that your declaring declaration includes an implementation. That implementation might be accessing the compiler synthesized backing field for that property. **CS9264** indicates that the your use of `field` assumes a non-nullable backing field while the property declaration is nullable. The compiler assumes both the backing field and the property have the same nullability. You need to add the `[field:MaybeNull, AllowNull]` attribute to the property declaration to indicate that the `field` value should be considered nullable. **CS9266** indicates that one of a properties accessors uses the `field` keyword, but the other uses a hand-declared backing field. The warning indicates you may have done that by accident.

Errors and warnings related to the `params` modifier on method parameters

Article • 02/25/2025

There are a few *errors* related to the `lock` statement and thread synchronization:

- CS0225: *The params parameter must be a single-dimensional array or have a valid collection type*
- CS0231: *A params parameter must be the last parameter in a formal parameter list.*
- CS0466: *'method1' should not have a params parameter since 'method2' does not*
- CS0674: *Do not use `System.ParamArrayAttribute` or `System.ParamArrayAttribute`/`System.Runtime.CompilerServices.ParamCollectionAttribute`. Use the `params` keyword instead.*
- CS0758: *Both partial method declarations must use a `params` parameter or neither may use a `params` parameter*
- CS1104: *A parameter array cannot be used with `this` modifier on an extension method.*
- CS1611: *The params parameter cannot be declared as in `ref` or `out`*
- CS1670: *`params` is not valid in this context*
- CS1751: *Cannot specify a default value for a parameter array.*
- CS9218: *The type arguments for method cannot be inferred from the usage because an argument with dynamic type is used and the method has a non-array params collection parameter. Try specifying the type arguments explicitly.*
- CS9223: *Creation of params collection results in an infinite chain of invocation of constructor.*
- CS9224: *Method cannot be less visible than the member with params collection.*
- CS9225: *Constructor leaves required member uninitialized.*
- CS9227: *Type does not contain a definition for a suitable instance `Add` method.*
- CS9228: *Non-array params collection type must have an applicable constructor that can be called with no arguments.*
- CS9272: *Implicitly typed lambda parameter cannot have the 'params' modifier.*

Method declaration rules

The following errors indicate using a `params` modifier on a parameter when the `params` modifier isn't allowed in that context:

- **CS0231**: *A params parameter must be the last parameter in a formal parameter list.*

- **CS1104**: *A parameter array cannot be used with `this` modifier on an extension method.*
- **CS1611**: *The params parameter cannot be declared as in `ref` or `out`*
- **CS1670**: *`params` is not valid in this context*
- **CS1751**: *Cannot specify a default value for a parameter array.*

The compiler enforces the following rules on your use of the `params` modifier on a method parameter:

- The `params` modifier is allowed only on the last parameter in a formal parameter list. This includes any parameters with a default value.
- You can't include a default argument for the parameter when the `params` modifier is used.
- The `params` modifier can't be applied to reference parameter. A reference parameter is one with the `in`, `ref readonly`, `ref` or `out` modifier.
- The `params` modifier can't be combined with the `this` modifier on an extension method.
- The `params` modifier can't be used on an overloaded operator.

In versions before C# 12, the `params` modifier can't be used on the parameter of an anonymous method or lambda expression.

Parameter and argument type rules

The following errors indicate that the type of the parameter used with `params` is invalid:

- **CS9218**: *The type arguments for method cannot be inferred from the usage because an argument with dynamic type is used and the method has a non-array params collection parameter. Try specifying the type arguments explicitly.*
- **CS0225**: *The params parameter must be a single-dimensional array or have a valid collection type*
- **CS9227**: *Type does not contain a definition for a suitable instance `Add` method.*
- **CS9228**: *Non-array params collection type must have an applicable constructor that can be called with no arguments.*

In versions before C# 13, the `params` modifier is allowed on single-dimensional arrays only. No other types were valid.

Starting with C# 13 any valid collection type can be used. However, some restrictions remain. The collection type must follow the same rules as the target of a collection expression.

Other params errors

The following errors indicate other issues with using the `params` modifier:

- **CS0466**: *'method1' should not have a params parameter since 'method2' does not*
- **CS0674**: *Do not use `System.ParamArrayAttribute` or `System.Runtime.CompilerServices.ParamCollectionAttribute`. Use the `params` keyword instead.*
- **CS0758**: *Both partial method declarations must use a `params` parameter or neither may use a `params` parameter*
- **CS9223**: *Creation of params collection results in an infinite chain of invocation of constructor.*
- **CS9224**: *Method cannot be less visible than the member with params collection.*
- **CS9225**: *Constructor leaves required member uninitialized.*
- **CS9272**: *Implicitly typed lambda parameter cannot have the 'params' modifier.*

A method that implements an interface must include the `params` modifier if and only if the interface member has the `params` modifier. Similarly, either both declarations of a `partial` method must include the `params` modifier, or none can include the `params` modifier.

You must use the `params` modifier. You can't apply the equivalent attributes, either System.ParamArrayAttribute or System.Runtime.CompilerServices.ParamCollectionAttribute.

The compiler generates one of the final three errors in the preceding list when the code generated to create the collection type is invalid:

- The compiler emits **CS9223** when the code emitted to create the collection also contains a params collection of the same type. Typically, the `Add` method takes a `params` collection of the same type.
- The compiler emits **CS9224** when the `Create` method for the collection type is less accessible than the method that takes the `params` parameter of the collection type.
- The compiler emits **CS9225** when the collection type has a required member and the parameterless constructor doesn't initialize that member and have the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute on the parameterless constructor.
- The compiler emits **CS9272** when you've used the `params` modifier without type information on a lambda expression. You must specify the types for all lambda expression parameters to use the `params` modifier.

See also

- Extension Methods
- Partial Classes and Methods
- Collection expressions
- params

Resolve nullable warnings

Article • 02/22/2025

The purpose of nullable warnings is to minimize the chance that your application throws a System.NullReferenceException when run. To achieve this goal, the compiler uses static analysis and issues warnings when your code has constructs that might lead to null reference exceptions. You provide the compiler with information for its static analysis by applying type annotations and attributes. These annotations and attributes describe the nullability of arguments, parameters, and members of your types. In this article, you learn different techniques to address the nullable warnings the compiler generates from its static analysis. The techniques described here are for general C# code. Learn to work with nullable reference types and Entity Framework core in Working with nullable reference types.

Nullable reference types, including the operators `?` and `!` are allowed only when the nullable context is set to `enable` or `annotations`. You can set the nullable context using the `Nullable` compiler option in your project file, or using the #nullable pragma in your source code.

This article covers the following compiler warnings:

- **CS8597** - *Thrown value may be null.*
- **CS8598** - *The suppression operator is not allowed in this context*
- **CS8600** - *Converting null literal or possible null value to non-nullable type.*
- **CS8601** - *Possible null reference assignment.*
- **CS8602** - *Dereference of a possibly null reference.*
- **CS8603** - *Possible null reference return.*
- **CS8604** - *Possible null reference argument for parameter.*
- **CS8605** - *Unboxing a possibly null value.*
- **CS8607** - *A possible null value may not be used for a type marked with `[NotNull]` or `[DisallowNull]`*
- **CS8608** - *Nullability of reference types in type doesn't match overridden member.*
- **CS8609** - *Nullability of reference types in return type doesn't match overridden member.*
- **CS8610** - *Nullability of reference types in type parameter doesn't match overridden member.*
- **CS8611** - *Nullability of reference types in type parameter doesn't match partial method declaration.*
- **CS8612** - *Nullability of reference types in type doesn't match implicitly implemented member.*

- CS8613 - *Nullability of reference types in return type doesn't match implicitly implemented member.*
- CS8614 - *Nullability of reference types in type of parameter doesn't match implicitly implemented member.*
- CS8615 - *Nullability of reference types in type doesn't match implemented member.*
- CS8616 - *Nullability of reference types in return type doesn't match implemented member.*
- CS8617 - *Nullability of reference types in type of parameter doesn't match implemented member.*
- CS8618 - *Non-nullable variable must contain a non-null value when exiting constructor. Consider declaring it as nullable.*
- CS8619 - *Nullability of reference types in value doesn't match target type.*
- CS8620 - *Argument cannot be used for parameter due to differences in the nullability of reference types.*
- CS8621 - *Nullability of reference types in return type doesn't match the target delegate (possibly because of nullability attributes).*
- CS8622 - *Nullability of reference types in type of parameter doesn't match the target delegate (possibly because of nullability attributes).*
- CS8623 - *Explicit application of* `System.Runtime.CompilerServices.NullableAttribute` *is not allowed.*
- CS8624 - *Argument cannot be used as an output due to differences in the nullability of reference types.*
- CS8625 - *Cannot convert null literal to non-nullable reference type.*
- CS8628 - *Cannot use a nullable reference type in object creation.*
- CS8629 - *Nullable value type may be null.*
- CS8631 - *The type cannot be used as type parameter in the generic type or method. Nullability of type argument doesn't match constraint type.*
- CS8632 - *The annotation for nullable reference types should only be used in code within a* `#nullable` *annotations context.*
- CS8633 - *Nullability in constraints for type parameter of method doesn't match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- CS8634 - *The type cannot be used as type parameter in the generic type or method. Nullability of type argument doesn't match 'class' constraint.*
- CS8636 - *Invalid option for* `/nullable`*; must be* `disable`, `enable`, `warnings` *or* `annotations`
- CS8637 - *Expected* `enable`, `disable`, *or* `restore`
- CS8639 - *The typeof operator cannot be used on a nullable reference type*
- CS8643 - *Nullability of reference types in explicit interface specifier doesn't match interface implemented by the type.*

- CS8644 - *Type does not implement interface member. Nullability of reference types in interface implemented by the base type doesn't match.*
- CS8645 - *Member is already listed in the interface list on type with different nullability of reference types.*
- CS8655 - *The switch expression does not handle some null inputs (it is not exhaustive).*
- CS8667 - *Partial method declarations have inconsistent nullability in constraints for type parameter.*
- CS8670 - *Object or collection initializer implicitly dereferences possibly null member.*
- CS8714 - *The type cannot be used as type parameter in the generic type or method. Nullability of type argument doesn't match 'notnull' constraint.*
- CS8762 - *Parameter must have a non-null value when exiting.*
- CS8763 - *A method marked `[DoesNotReturn]` should not return.*
- CS8764 - *Nullability of return type doesn't match overridden member (possibly because of nullability attributes).*
- CS8765 - *Nullability of type of parameter doesn't match overridden member (possibly because of nullability attributes).*
- CS8766 - *Nullability of reference types in return type of doesn't match implicitly implemented member (possibly because of nullability attributes).*
- CS8767 - *Nullability of reference types in type of parameter of doesn't match implicitly implemented member (possibly because of nullability attributes).*
- CS8768 - *Nullability of reference types in return type doesn't match implemented member (possibly because of nullability attributes).*
- CS8769 - *Nullability of reference types in type of parameter doesn't match implemented member (possibly because of nullability attributes).*
- CS8770 - *Method lacks `[DoesNotReturn]` annotation to match implemented or overridden member.*
- CS8774 - *Member must have a non-null value when exiting.*
- CS8776 - *Member cannot be used in this attribute.*
- CS8775 - *Member must have a non-null value when exiting.*
- CS8777 - *Parameter must have a non-null value when exiting.*
- CS8819 - *Nullability of reference types in return type doesn't match partial method declaration.*
- CS8824 - *Parameter must have a non-null value when exiting because parameter is non-null.*
- CS8825 - *Return value must be non-null because parameter is non-null.*
- CS8847 - *The switch expression does not handle some null inputs (it is not exhaustive). However, a pattern with a 'when' clause might successfully match this value.*

> **Note**
>
> The static analysis can't always deduce in what order, in a specific scenario, methods are accessed, and whether the method completes successfully without throwing an exception. Those known pitfalls are well described in Known pitfalls section.

You address almost all warnings using one of five techniques:

- Configuring the nullable context.
- Adding necessary null checks.
- Adding or removing `?` or `!` nullable annotations.
- Adding attributes that describe null semantics.
- Initializing variables correctly.

If you're new to using nullable reference types, the overview of nullable reference types provides a background on what nullable reference types solve and how they work to provide warnings to possible mistakes in your code. You can also check the guidance on migrating to nullable reference types to learn more about enabling nullable reference types in an existing project.

Configure nullable context

The following warnings indicate that you haven't set the nullable context correctly:

- **CS8632** - *The annotation for nullable reference types should only be used in code within a `#nullable` annotations context.*
- **CS8636** - *Invalid option for `/nullable`; must be `disable`, `enable`, `warnings` or `annotations`*
- **CS8637** - *Expected `enable`, `disable`, or `restore`*

Incorrect annotation syntax

These errors and warnings indicate that usage of the `!` or `?` annotation is incorrect.

- **CS8598** - *The suppression operator is not allowed in this context*
- **CS8623** - *Explicit application of `System.Runtime.CompilerServices.NullableAttribute` is not allowed.*
- **CS8628** - *Cannot use a nullable reference type in object creation.*
- **CS8639** - *The typeof operator cannot be used on a nullable reference type*

The `?` annotation in a declaration indicates that the variable might be null. It doesn't indicate a different runtime type. Both the following declarations are the same runtime type:

C#

```
string s1 = "a string";
string? s2 = "another string";
```

The `?` is a hint to the compiler on the expectation for null values.

The `!` annotation on an expression indicates that you know the expression is safe and should be assumed to be not null.

- You must use these annotations, not the System.Runtime.CompilerServices.NullableAttribute in your code.
- Because the `?` is an annotation, not a type, you can't use it with typeof, or new expressions.
- The `!` operator can't be applied to a variable expression or a method group.
- The `!` operator can't be applied to the left of a member access operator, such as `obj.Field!.Method()`.

Possible dereference of null

This set of warnings alerts you that you're dereferencing a variable whose *null-state* is *maybe-null*. These warnings are:

- **CS8602** - *Dereference of a possibly null reference.*
- **CS8670** - *Object or collection initializer implicitly dereferences possibly null member.*

The following code demonstrates one example of each of the preceding warnings:

C#

```
class Container
{
    public List<string>? States { get; set; }
}

internal void PossibleDereferenceNullExamples(string? message)
{
    Console.WriteLine(message.Length); // CS8602

    var c = new Container { States = { "Red", "Yellow", "Green" } }; //
```

```
CS8670
}
```

In the preceding example, the warning is because the `Container`, `c`, might have a null value for the `States` property. Assigning new states to a collection that might be null causes the warning.

To remove these warnings, you need to add code to change that variable's *null-state* to *not-null* before dereferencing it. The collection initializer warning can be harder to spot. The compiler detects that the collection *maybe-null* when the initializer adds elements to it.

In many instances, you can fix these warnings by checking that a variable isn't null before dereferencing it. Consider the following example that adds a null check before dereferencing the `message` parameter:

C#

```
void WriteMessageLength(string? message)
{
    if (message is not null)
    {
        Console.WriteLine(message.Length);
    }

}
```

The following example initializes the backing storage for the `States` and removes the `set` accessor. Consumers of the class can modify the contents of the collection, and the storage for the collection is never `null`:

C#

```
class Container
{
    public List<string> States { get; } = new();
}
```

Other instances when you get these warnings might be false positive. You might have a private utility method that tests for null. The compiler doesn't know that the method provides a null check. Consider the following example that uses a private utility method, `IsNotNull`:

C#

```
public void WriteMessage(string? message)
{
    if (IsNotNull(message))
        Console.WriteLine(message.Length);
}
```

The compiler warns that you might be dereferencing null when you write the property `message.Length` because its static analysis determines that `message` might be `null`. You know that `IsNotNull` provides a null check, and when it returns `true`, the *null-state* of `message` should be *not-null*. You must tell the compiler those facts. One way is to use the null forgiving operator, `!`. You can change the `WriteLine` statement to match the following code:

C#

```
Console.WriteLine(message!.Length);
```

The null forgiving operator makes the expression *not-null* even if it was *maybe-null* without the `!` applied. In this example, a better solution is to add an attribute to the signature of `IsNotNull`:

C#

```
private static bool IsNotNull([NotNullWhen(true)] object? obj) => obj !=
null;
```

The System.Diagnostics.CodeAnalysis.NotNullWhenAttribute informs the compiler that the argument used for the `obj` parameter is *not-null* when the method returns `true`. When the method returns `false`, the argument has the same *null-state* it had before the method was called.

> **Tip**
>
> There's a rich set of attributes you can use to describe how your methods and properties affect *null-state*. You can learn about them in the language reference article on **Nullable static analysis attributes**.

Fixing a warning for dereferencing a *maybe-null* variable involves one of three techniques:

- Add a missing null check.

- Add null analysis attributes on APIs to affect the compiler's *null-state* static analysis. These attributes inform the compiler when a return value or argument should be *maybe-null* or *not-null* after calling the method.
- Apply the null forgiving operator `!` to the expression which forces the state to *not-null*.

Possible null assigned to a nonnullable reference

This set of warnings alerts you that you're assigning a variable whose type is nonnullable to an expression whose *null-state* is *maybe-null*. These warnings are:

- **CS8597** - *Thrown value may be null.*
- **CS8600** - *Converting null literal or possible null value to non-nullable type.*
- **CS8601** - *Possible null reference assignment.*
- **CS8603** - *Possible null reference return.*
- **CS8604** - *Possible null reference argument for parameter.*
- **CS8605** - *Unboxing a possibly null value.*
- **CS8625** - *Cannot convert null literal to non-nullable reference type.*
- **CS8629** - *Nullable value type may be null.*

The compiler emits these warnings when you attempt to assign an expression that is *maybe-null* to a variable that is nonnullable. For example:

C#

```
string? TryGetMessage(int id) => "";

string msg = TryGetMessage(42);  // Possible null assignment.
```

The different warnings indicate provide details about the code, such as assignment, unboxing assignment, return statements, arguments to methods, and throw expressions.

You can take one of three actions to address these warnings. One is to add the `?` annotation to make the variable a nullable reference type. That change can cause other warnings. Changing a variable from a non-nullable reference to a nullable reference changes its default *null-state* from *not-null* to *maybe-null*. The compiler's static analysis finds instances where you dereference a variable that is *maybe-null*.

The other actions instruct the compiler that the right-hand-side of the assignment is *not-null*. The expression on the right-hand-side could be null-checked before assignment, as shown in the following example:

C#

```
string notNullMsg = TryGetMessage(42) ?? "Unknown message id: 42";
```

The previous examples demonstrate assignment of the return value of a method. You annotate the method (or property) to indicate when a method returns a not-null value. The System.Diagnostics.CodeAnalysis.NotNullIfNotNullAttribute often specifies that a return value is *not-null* when an input argument is *not-null*. Another alternative is to add the null forgiving operator, `!` to the right-hand side:

C#

```
string msg = TryGetMessage(42)!;
```

Fixing a warning for assigning a *maybe-null* expression to a *not-null* variable involves one of four techniques:

- Change the left side of the assignment to a nullable type. This action can introduce new warnings when you dereference that variable.
- Provide a null-check before the assignment.
- Annotate the API that produces the right-hand side of the assignment.
- Add the null forgiving operator to the right-hand side of the assignment.

Nonnullable reference not initialized

This set of warnings alerts you that you're assigning a variable whose type is non-nullable to an expression whose *null-state* is *maybe-null*. These warnings are:

- **CS8618** - *Non-nullable variable must contain a non-null value when exiting constructor. Consider declaring it as nullable.*
- **CS8762** - *Parameter must have a non-null value when exiting.*

Consider the following class as an example:

C#

```
public class Person
{
    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

Neither `FirstName` nor `LastName` are guaranteed initialized. If this code is new, consider changing the public interface. The preceding example could be updated as follows:

C#

```
public class Person
{
    public Person(string first, string last)
    {
        FirstName = first;
        LastName = last;
    }

    public string FirstName { get; set; }
    public string LastName { get; set; }
}
```

If you require creating a `Person` object before setting the name, you can initialize the properties using a default non-null value:

C#

```
public class Person
{
    public string FirstName { get; set; } = string.Empty;
    public string LastName { get; set; } = string.Empty;
}
```

Another alternative is to change those members to nullable reference types. The `Person` class could be defined as follows if `null` should be allowed for the name:

C#

```
public class Person
{
    public string? FirstName { get; set; }
    public string? LastName { get; set; }
}
```

Existing code sometimes require other changes to inform the compiler about the null semantics for those members. It might have multiple constructors, and your class has a private helper method that initializes one or more members. You can move the initialization code into a single constructor and ensure all constructors call the one with the common initialization code. Or, you can use the System.Diagnostics.CodeAnalysis.MemberNotNullAttribute and System.Diagnostics.CodeAnalysis.MemberNotNullWhenAttribute attributes. These

attributes inform the compiler that a member is *not-null* after the method returns. 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";
    }
}
```

Finally, you can use the null forgiving operator to indicate that a member is initialized in other code. For another example, consider the following classes representing an Entity Framework Core model:

```
C#
```

```
public class TodoItem
{
    public long Id { get; set; }
    public string? Name { get; set; }
    public bool IsComplete { get; set; }
}

public class TodoContext : DbContext
{
    public TodoContext(DbContextOptions<TodoContext> options)
        : base(options)
    {
    }

    public DbSet<TodoItem> TodoItems { get; set; } = null!;
}
```

The `DbSet` property is initialized to `null!`. That tells the compiler that the property is set to a *not-null* value. In fact, the base `DbContext` performs the initialization of the set. The compiler's static analysis doesn't pick that up. For more information on working with nullable reference types and Entity Framework Core, see the article on Working with Nullable Reference Types in EF Core.

Fixing a warning for not initializing a nonnullable member involves one of four techniques:

- Change the constructors or field initializers to ensure all nonnullable members are initialized.
- Change one or more members to be nullable types.
- Annotate any helper methods to indicate which members are assigned.
- Add an initializer to `null!` to indicate that the member is initialized in other code.

Mismatch in nullability declaration

Many warnings indicate nullability mismatches between signatures for methods, delegates, or type parameters.

- **CS8608** - *Nullability of reference types in type doesn't match overridden member.*

- **CS8609** - *Nullability of reference types in return type doesn't match overridden member.*
- **CS8610** - *Nullability of reference types in type parameter doesn't match overridden member.*
- **CS8611** - *Nullability of reference types in type parameter doesn't match partial method declaration.*
- **CS8612** - *Nullability of reference types in type doesn't match implicitly implemented member.*
- **CS8613** - *Nullability of reference types in return type doesn't match implicitly implemented member.*
- **CS8614** - *Nullability of reference types in type of parameter doesn't match implicitly implemented member.*
- **CS8615** - *Nullability of reference types in type doesn't match implemented member.*
- **CS8616** - *Nullability of reference types in return type doesn't match implemented member.*
- **CS8617** - *Nullability of reference types in type of parameter doesn't match implemented member.*
- **CS8619** - *Nullability of reference types in value doesn't match target type.*
- **CS8620** - *Argument cannot be used for parameter due to differences in the nullability of reference types.*
- **CS8621** - *Nullability of reference types in return type doesn't match the target delegate (possibly because of nullability attributes).*
- **CS8622** - *Nullability of reference types in type of parameter doesn't match the target delegate (possibly because of nullability attributes).*
- **CS8624** - *Argument cannot be used as an output due to differences in the nullability of reference types.*
- **CS8631** - *The type cannot be used as type parameter in the generic type or method. Nullability of type argument doesn't match constraint type.*
- **CS8633** - *Nullability in constraints for type parameter of method doesn't match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- **CS8634** - *The type cannot be used as type parameter in the generic type or method. Nullability of type argument doesn't match 'class' constraint.*
- **CS8643** - *Nullability of reference types in explicit interface specifier doesn't match interface implemented by the type.*
- **CS8644** - *Type does not implement interface member. Nullability of reference types in interface implemented by the base type doesn't match.*
- **CS8645** - *Member is already listed in the interface list on type with different nullability of reference types.*
- **CS8667** - *Partial method declarations have inconsistent nullability in constraints for type parameter.*

- **CS8714** - *The type cannot be used as type parameter in the generic type or method. Nullability of type argument doesn't match 'notnull' constraint.*
- **CS8764** - *Nullability of return type doesn't match overridden member (possibly because of nullability attributes).*
- **CS8765** - *Nullability of type of parameter doesn't match overridden member (possibly because of nullability attributes).*
- **CS8766** - *Nullability of reference types in return type of doesn't match implicitly implemented member (possibly because of nullability attributes).*
- **CS8767** - *Nullability of reference types in type of parameter of doesn't match implicitly implemented member (possibly because of nullability attributes).*
- **CS8768** - *Nullability of reference types in return type doesn't match implemented member (possibly because of nullability attributes).*
- **CS8769** - *Nullability of reference types in type of parameter doesn't match implemented member (possibly because of nullability attributes).*
- **CS8819** - *Nullability of reference types in return type doesn't match partial method declaration.*

The following code demonstrates *CS8764*:

C#

```
public class B
{
    public virtual string GetMessage(string id) => string.Empty;
}
public class D : B
{
    public override string? GetMessage(string? id) => default;
}
```

The preceding example shows a `virtual` method in a base class and an `override` with different nullability. The base class returns a non-nullable string, but the derived class returns a nullable string. If the `string` and `string?` are reversed, it would be allowed because the derived class is more restrictive. Similarly, parameter declarations should match. Parameters in the override method can allow null even when the base class doesn't.

Other situations can generate these warnings. You have a mismatch in an interface method declaration and the implementation of that method. Or a delegate type and the expression for that delegate differ. A type parameter and the type argument differ in nullability.

To fix these warnings, update the appropriate declaration.

Code doesn't match attribute declaration

The preceding sections discussed how you can use Attributes for nullable static analysis to inform the compiler about the null semantics of your code. The compiler warns you if the code doesn't adhere to the promises of that attribute:

- **CS8607** - *A possible null value may not be used for a type marked with* `[NotNull]` *or* `[DisallowNull]`
- **CS8763** - *A method marked* `[DoesNotReturn]` *should not return.*
- **CS8770** - *Method lacks* `[DoesNotReturn]` *annotation to match implemented or overridden member.*
- **CS8774** - *Member must have a non-null value when exiting.*
- **CS8775** - *Member must have a non-null value when exiting.*
- **CS8776** - *Member cannot be used in this attribute.*
- **CS8777** - *Parameter must have a non-null value when exiting.*
- **CS8824** - *Parameter must have a non-null value when exiting because parameter is non-null.*
- **CS8825** - *Return value must be non-null because parameter is non-null.*

Consider the following method:

C#

```
public bool TryGetMessage(int id, [NotNullWhen(true)] out string? message)
{
    message = null;
    return true;

}
```

The compiler produces a warning because the `message` parameter is assigned `null` *and* the method returns `true`. The `NotNullWhen` attribute indicates that shouldn't happen.

To address these warnings, update your code so it matches the expectations of the attributes applied. You can change the attributes, or the algorithm.

Exhaustive switch expression

Switch expressions must be *exhaustive*, meaning that all input values must be handled. Even for non-nullable reference types, the `null` value must be accounted for. The compiler issues warnings when the null value isn't handled:

- **CS8655** - *The switch expression does not handle some null inputs (it is not exhaustive).*
- **CS8847** - *The switch expression does not handle some null inputs (it is not exhaustive). However, a pattern with a 'when' clause might successfully match this value.*

The following example code demonstrates this condition:

```C#
int AsScale(string status) =>
    status switch
    {
        "Red" => 0,
        "Yellow" => 5,
        "Green" => 10,
        { } => -1
    };
```

The input expression is a `string`, not a `string?`. The compiler still generates this warning. The `{ }` pattern handles all non-null values, but doesn't match `null`. To address these errors, you can either add an explicit `null` case, or replace the `{ }` with the `_` (discard) pattern. The discard pattern matches null in addition to any other value.

Resolve errors and warnings in pattern matching expressions

This article covers the following compiler errors and warnings:

- CS8509: *The switch expression does not handle all possible values of its input type (it is not exhaustive). For example, the pattern '...' is not covered.*
- CS9134: *A switch expression arm does not begin with a 'case' keyword.*
- CS9135: *A constant value of type is expected*
- CS9335, CS9336: *The pattern is redundant.*
- CS9337: *The pattern is too complex to analyze for redundancy.*

Switch expression syntax errors

- **CS9134**: *A switch expression arm does not begin with a 'case' keyword.*
- **CS9135**: *A constant value of type is expected*

To write switch expressions correctly, follow the proper syntax rules. For more information, see Switch expression.

- Don't use the `case` keyword in switch expressions (**CS9134**). The `case` keyword is used in switch statements, not switch expressions. In a switch expression, each arm consists of a pattern, an optional case guard, the `=>` token, and an expression.
- Use constant values in patterns, not variables (**CS9135**). Pattern matching requires compile-time constants for value patterns.

The following example demonstrates the syntax error with `case`:

C#

```
var answer = x switch
{
    // CS9134: remove the keyword "case" from each switch arm:
    case 0 => false,
    case 1 => true,
}
```

To fix this error, remove the `case` keyword:

C#

```
var answer = x switch
{
    0 => false,
```

```
    1 => true,
}
```

Incomplete pattern matching

- **CS8509**: *The switch expression does not handle all possible values of its input type (it is not exhaustive). For example, the pattern '...' is not covered.*

To create exhaustive switch expressions, cover all possible input values. For more information, see Switch expression and Switch statement.

Add switch arms that handle all possible input values. Use the discard pattern (`_`) to match any remaining values that you don't need to handle explicitly.

The following example generates CS8509:

C#

```
// CS8509.cs
enum EnumValues
{
    Value1,
    Value2,
    Value3
}

void Method(EnumValues enumValues)
{
    var result = enumValues switch
    {
        EnumValues.Value1 => 1,
        EnumValues.Value2 => 2,
    };
}
```

To fix this warning, add a default arm:

C#

```
enum EnumValues
{
    Value1,
    Value2,
    Value3
}

void Method(EnumValues enumValues)
{
    var result = enumValues switch
```

```
    {
        EnumValues.Value1 => 1,
        EnumValues.Value2 => 2,
        EnumValues.Value3 => 3,
        _ => throw new ArgumentException("Input isn't a valid enum value",
nameof(enumValues)),
        };
}
```

The `_` pattern matches all remaining values. You can use this pattern to handle invalid or unexpected values.

Redundant patterns

- **CS9335, CS9336**: *The pattern is redundant.*
- **CS9337**: *The pattern is too complex to analyze for redundancy.*

To write clear pattern matching expressions, avoid redundant patterns. For more information, see Patterns.

- Review patterns that the compiler identifies as redundant (**CS9335**, **CS9336**). Redundant patterns can indicate a logic error where you meant to use `not` or different logical operators.
- Simplify complex patterns that are too difficult for the compiler to analyze (**CS9337**). Break down complex patterns into simpler, more maintainable expressions.

The following example demonstrates a redundant pattern:

C#

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

This warning indicates you likely meant `is not (null or 42)` or `is not (int or string)` instead of using `or` at the top level.

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

---

Last updated on 11/08/2025

Errors and warnings for string literal declarations

10/11/2025

There are several errors related to declaring string constants or string literals.

- CS1009: *Unrecognized escape sequence.*
- CS1010: *Newline in constant.*
- CS1011: *Empty character literal.*
- CS1012: *Too many characters in character literal.*
- CS1039: *Unterminated string literal.*
- CS8997: *Unterminated raw string literal.*
- CS8998: *Not enough starting quotes for this raw string content.*
- CS8999: *Line does not start with the same whitespace as the closing line of the raw string literal.*
- CS9000: *Raw string literal delimiter must be on its own line.*
- CS9001: *Multi-line raw string literals are only allowed in verbatim interpolated strings.*
- CS9002: *Multi-line raw string literals must contain at least one line of content.*
- CS9003: *Line contains different whitespace than expected.*
- CS9004: *Not enough quotes for raw string literal.*
- CS9005: *Not enough closing braces for interpolated raw string literal.*
- CS9006: *Too many opening braces for interpolated raw string literal.*
- CS9007: *Too many closing braces for interpolated raw string literal.*
- CS9008: *Sequence of '@' characters is not allowed.*
- CS9009: *String must start with quote character.*
- CS9274: *Cannot emit this string literal into the data section because it has XXHash128 collision with another string literal.*
- CS9315: *Combined length of user strings used by the program exceeds allowed limit. Adding a string literal requires restarting the application.*

The following sections provide examples of common issues and how to fix them.

Incorrectly formed string literals

The following errors concern string and character literal syntax and common mistakes when declaring literal values.

- **CS1009** - *Unrecognized escape sequence.*
- **CS1010** - *Newline in constant.*
- **CS1011** - *Empty character literal.*

- **CS1012** - *Too many characters in character literal.*
- **CS1039** - *Unterminated string literal.*

Common causes and fixes:

- Invalid escape sequences: An unexpected character follows a backslash (`\\`). Use valid escapes (`\\n`, `\\t`, `\\uXXXX`, `\\xX`) or use verbatim (`@"..."`) or raw string literals for content that includes backslashes.
- Empty or multi-character char literals: Character literals must contain exactly one UTF-16 code unit. Use a single character like `'x'` or a string / `System.Text.Rune` for characters outside the BMP.
- Unterminated strings: Ensure every string or verbatim string has a matching closing quote. For verbatim strings, the final `"` must be present; for normal strings ensure escaped quotes are balanced.
- A string literal spans multiple lines of C# source.

Examples

C#

```
// CS1009 - invalid escape
string a = "\\m";               // CS1009 - invalid escape \m

// Use verbatim strings or escape backslashes
string filename = "c:\\myFolder\\myFile.txt"; // escaped backslashes
string filenameVerbatim = @"c:\myFolder\myFile.txt"; // verbatim string

// CS1011 - empty character literal
// public char CharField = '';   // CS1011 - invalid: empty character literal

// CS1012 - too many characters in char literal
char a = 'xx';   // CS1012 - too many characters

// CS1039 - unterminated verbatim string
// string b = @"hello, world;   // CS1039 - missing closing quote
```

For more information on literal strings and escape sequences, see the articles on verbatim strings and raw strings.

Incorrectly formed raw string literals

The following errors are related to raw string literal syntax and usage.

- **CS8997** - *Unterminated raw string literal.*
- **CS8998** - *Not enough starting quotes for this raw string content.*

- **CS8999** - *Line does not start with the same whitespace as the closing line of the raw string literal.*
- **CS9000** - *Raw string literal delimiter must be on its own line.*
- **CS9001** - *Multi-line raw string literals are only allowed in verbatim interpolated strings.*
- **CS9002** - *Multi-line raw string literals must contain at least one line of content.*
- **CS9003** - *Line contains different whitespace than expected.*
- **CS9004** - *Not enough quotes for raw string literal.*
- **CS9005** - *Not enough closing braces for interpolated raw string literal.*
- **CS9006** - *Too many opening braces for interpolated raw string literal.*
- **CS9007** - *Too many closing braces for interpolated raw string literal.*
- **CS9008** - *Sequence of '@' characters is not allowed.*
- **CS9009** - *String must start with quote character.*

Check these common causes and fixes:

- Unterminated or mismatched delimiters: Ensure your raw string starts and ends with the same number of consecutive double quotes (`"`). For multi-line raw strings, the opening and closing delimiter lines must appear on their own lines.
- Indentation and whitespace mismatch: The indentation of the closing delimiter defines the trimming of common leading whitespace for content lines. Make sure content lines align with that indentation.
- Insufficient quote or `$` counts for content: If the content begins with runs of quote characters or brace characters, increase the length of the delimiter (more `"`) or the number of leading `$` characters for interpolated raw strings so content can't be confused with delimiters or interpolation.
- Illegal characters or sequences: Avoid multiple `@` characters for verbatim/raw combinations and ensure you use verbatim interpolated forms when combining interpolation with multi-line raw strings.

The following code shows a few examples of incorrectly formed raw string literals.

C#

```
// Unterminated raw string (CS8997)
var s = """This raw string never ends...

// Delimiter must be on its own line (CS9000)
var t = """First line
           More text
         """;
```

For full syntax and more examples, see the language reference on raw string literals.

Literal strings in data sections

- **CS9274**: *Cannot emit this string literal into the data section because it has XXHash128 collision with another string literal.*
- **CS9315**: *Combined length of user strings used by the program exceeds allowed limit. Adding a string literal requires restarting the application.*

CS9274 indicate that your declaration can't be emitted in the data section. Disable this feature for your application. Debugging tools emit **CS9315** after you changed string data in the data section while debugging and your app must be restarted.

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

Resolve errors and warnings in array and collection declarations and initialization expressions

This article covers the following compiler errors:

- CS0022: *Wrong number of indices inside [], expected 'number'*
- CS0178: *Invalid rank specifier: expected '`,`' or '`]`'*
- CS0248: *Cannot create an array with a negative size*
- CS0270: *Array size cannot be specified in a variable declaration (try initializing with a '`new`' expression)*
- CS0611: *Array elements cannot be of type*
- CS0623: *Array initializers can only be used in a variable or field initializer. Try using a new expression instead.*
- CS0650: *Bad array declarator: To declare a managed array the rank specifier precedes the variable's identifier. To declare a fixed size buffer field, use the fixed keyword before the field type.*
- CS0719: *Array elements cannot be of static type*
- CS0747: *Invalid initializer member declarator.*
- CS0820: *Cannot assign array initializer to an implicitly typed local*
- CS0826: *No best type found for implicitly typed array.*
- CS0846: *A nested array initializer is expected*
- CS1063: *The best overloaded Add method for the collection initializer element is obsolete.*
- CS1552: *Array type specifier, `[]`, must appear before parameter name*
- CS1586: *Array creation must have array size or array initializer*
- CS1920: *Element initializer cannot be empty.*
- CS1921: *The best overloaded method match has wrong signature for the initializer element. The initializable `Add` must be an accessible instance method.*
- CS1925: *Cannot initialize object of type 'type' with a collection initializer.*
- CS1950: *The best overloaded Add method for the collection initializer has some invalid arguments.*
- CS1954: *The best overloaded method match for the collection initializer element cannot be used. Collection initializer '`Add`' methods cannot have `ref` or `out` parameters.*
- CS9174: *Cannot initialize type with a collection literal because the type is not constructible.*
- CS9176: *There is no target type for the collection literal.*
- CS9185: *The CollectionBuilderAttribute builder type must be a non-generic class or struct.*
- CS9186: *The CollectionBuilderAttribute method name is invalid.*
- CS9187: *Could not find an accessible method with the expected signature: a static method with a single parameter of type `ReadOnlySpan<T>`; and correct return type*

- CS9188: *Type has a CollectionBuilderAttribute but no element type.*
- CS9203: *A collection expression of this type cannot be used in this context because it may be exposed outside of the current scope.*
- CS9210: *This version of System.Collections.Immutable.ImmutableArray<T>cannot be used with collection expressions.*
- CS9212: *Spread operator '`..`' cannot operate on variables of type 'type' because 'type' does not contain a public instance or extension definition for 'member'.*
- CS9213: *Collection expression target 'type' has no element type.*
- CS9214: *Collection expression type must have an applicable constructor that can be called with no arguments.*
- CS9215: *Collection expression type 'type' must have an instance or extension method 'Add' that can be called with a single argument.*
- CS9222: *Collection initializer results in an infinite chain of instantiations of collection 'type'.*
- CS9332: *Cannot use '..' spread operator in the filter expression of a catch clause.*

In addition, the following warnings are covered in this article:

- CS1062: *The best overloaded Add method for the collection initializer element is obsolete.*
- CS1064: *The best overloaded Add method for the collection initializer element is obsolete.*
- CS3007: *Overloaded method 'method' differing only by unnamed array types is not CLS-compliant*
- CS3016: *Arrays as attribute arguments is not CLS-compliant*
- CS0251: *Indexing an array with a negative index (array indices always start at zero)*
- CS9208: *Collection expression may incur unexpected heap allocations. Consider explicitly creating an array, then converting to the final type to make the allocation explicit.*
- CS9209: *Collection expression may incur unexpected heap allocations due to use of '`..`' spreads. Consider explicitly creating an array, then converting to the final type to make the allocation explicit.*
- CS9332: *Cannot use '..' spread operator in the filter expression of a catch clause.*

Invalid array element access

- **CS0022**: *Wrong number of indices inside [], expected 'number'*
- **CS0251**: *Indexing an array with a negative index (array indices always start at zero)*

To access array elements correctly, follow these indexing rules. For more information, see Arrays.

- Specify the same number of indices as declared dimensions (**CS0022**). A one-dimensional array needs one index; a three-dimensional array needs three indices.
- Use only non-negative integers for array indices (**CS0251**). Array indices always start at zero.

Invalid collection initializer

- **CS0747**: *Invalid initializer member declarator.*
- **CS1920**: *Element initializer cannot be empty.*
- **CS1921**: *The best overloaded method match has wrong signature for the initializer element. The initializable `Add` must be an accessible instance method.*
- **CS1922**: *Cannot initialize type 'type' with a collection initializer because 'type' doesn't implement 'System.Collections.IEnumerable'.*
- **CS1925**: *Cannot initialize object of type 'type' with a collection initializer.*
- **CS1927**: *Warning: Ignoring /win32manifest for module because it only applies to assemblies*
- **CS1950**: *The best overloaded Add method for the collection initializer has some invalid arguments.*
- **CS1954**: *The best overloaded method match for the collection initializer element cannot be used. Collection initializer '`Add`' methods cannot have `ref` or `out` parameters.*
- **CS9174**: *Cannot initialize type with a collection literal because the type is not constructible.*
- **CS9176**: *There is no target type for the collection literal.*
- **CS9203**: *A collection expression of this type cannot be used in this context because it may be exposed outside of the current scope.*
- **CS9210**: *This version of System.Collections.Immutable.ImmutableArray<T> can't be used with collection expressions.*
- **CS9212**: *Spread operator '`..`' cannot operate on variables of type 'type' because 'type' doesn't contain a public instance or extension definition for 'member'.*
- **CS9213**: *Collection expression target 'type' has no element type.*
- **CS9214**: *Collection expression type must have an applicable constructor that can be called with no arguments.*
- **CS9215**: *Collection expression type 'type' must have an instance or extension method 'Add' that can be called with a single argument.*
- **CS9222**: *Collection initializer results in an infinite chain of instantiations of collection 'type'.*
- **CS9332**: *Cannot use '..' spread operator in the filter expression of a catch clause.*

The compiler might also generate the following warnings:

- **CS1062**: *The best overloaded Add method for the collection initializer element is obsolete.*
- **CS1063**: *The best overloaded Add method for the collection initializer element is obsolete.*
- **CS1064**: *The best overloaded Add method for the collection initializer element is obsolete.*
- **CS9208**: *Collection expression may incur unexpected heap allocations. Consider explicitly creating an array, then converting to the final type to make the allocation explicit.*
- **CS9209**: *Collection expression may incur unexpected heap allocations due to use of '`..`' spreads. Consider explicitly creating an array, then converting to the final type to make the allocation explicit.*

To create valid collection initializers, follow these rules. For more information, see Collection expressions.

- Don't mix property initialization with element addition in the same initializer (**CS0747**).
- Include at least one element in collection initializers with braces (**CS1920**).
- Ensure the collection type implements `IEnumerable` (**CS1922**).
- Use collection initializers only with collection types (**CS1925**).
- Verify the `Add` method is accessible, takes one parameter matching the element type, and doesn't use `ref` or `out` modifiers (**CS1921**, **CS1954**).
- Resolve ambiguous `Add` method overloads (**CS1950**).
- Provide an explicit target type for collection expressions when the compiler can't infer it (**CS9176**, **CS9213**).
- Ensure the collection type is constructible with a parameterless constructor (**CS9174**, **CS9214**).
- Don't use `ref struct` types in collection expressions that may violate ref safety (**CS9203**).
- Update to a compatible runtime version for `ImmutableArray` collection expressions (**CS9210**).
- Implement enumeration patterns (like `GetEnumerator`) for spread operator support (**CS9212**).
- Avoid circular dependencies in collection initialization (**CS9222**).
- Don't use the spread operator in catch clause filter expressions (**CS9332**).

Invalid array rank

- **CS0178**: *Invalid rank specifier: expected '`,`' or '`]`'*
- **CS0650**: *Bad array declarator: To declare a managed array the rank specifier precedes the variable's identifier. To declare a fixed size buffer field, use the fixed keyword before the field type.*
- **CS1552**: *Array type specifier, `[]`, must appear before parameter name*

To declare arrays correctly, follow the proper syntax order. For more information, see Arrays and the C# Language Specification section on array initializers.

An array declaration consists of these tokens in order:

1. The type of array elements (for example, `int`, `string`, or `SomeClassType`).
2. The array brackets, optionally including commas for multiple dimensions.
3. The variable name.

When specifying array dimensions, you can use:

- A number of elements in braces (`{` and `}`)

- Empty brackets
- One or more commas enclosed in brackets

The following examples show valid array declarations:

C#

```
int[] anArrayOfIntegers;
string[,,] threeDimensionalStringArray;
```

Invalid array length

- **CS0248**: *Cannot create an array with a negative size*
- **CS0270**: *Array size cannot be specified in a variable declaration (try initializing with a 'new' expression)*
- **CS1586**: *Array creation must have array size or array initializer*

To create arrays with valid lengths, specify the size during initialization, not declaration. For more information, see Arrays.

- Specify array length as part of initialization, not declaration (**CS0270**).
- Use only positive integers for array dimensions (**CS0248**).
- Provide either a size in a `new` expression or an array initializer (**CS1586**).

The following example shows both mechanisms:

C#

```
int[] fiveIntegers = new int[5];
string[] weekDays = { "Sun", "Mon", "Tue", "Wed", "Thu", "Fri", "Sat" };
```

Invalid element type

- **CS0611**: *Array elements cannot be of type 'type'*
- **CS0719**: *Array elements cannot be of static type*
- **CS0820**: *Cannot assign array initializer to an implicitly typed local*
- **CS0826**: *No best type found for implicitly typed array*

To use arrays with correct element types, follow these type restrictions. For more information, see Implicitly typed local variables and best common type.

- Don't use restricted types like System.TypedReference and System.ArgIterator as array element types (**CS0611**).

- Don't use `static` classes as array element types because instances can't be created (**CS0719**).
- Initialize implicitly typed arrays with a `new` expression (**CS0820**).
- Ensure all elements in an implicitly typed array initializer have a best common type (**CS0826**).

The following examples show how to declare implicitly typed arrays:

C#

```
var implicitType = new[] { 1, 2, 3 };

char c = 'c';
short s1 = 0;
short s2 = -0;
short s3 = 1;
short s4 = -1;

// common type is "int"
var commonType = new[] { s1, s2, s3, s4, c, 1 };
```

To ensure a best common type, use any of these techniques:

- Give the array an explicit type.
- Give all array elements the same type.
- Provide explicit casts on elements that might be causing the problem.

Invalid array initializer

- **CS0623**: *Array initializers can only be used in a variable or field initializer. Try using a new expression instead.*
- **CS0846**: *A nested array initializer is expected*
- **CS1925**: *Cannot initialize object of type 'type' with a collection initializer.*

These errors indicate invalid array initializer syntax. For more information, see Arrays.

To create valid array initializers:

- Use array initializers only in variable or field declarations (**CS0623**). Use a `new` expression in other contexts.
- Balance braces (`{` and `}`) around elements or child arrays (**CS0846**).
- Ensure the initializing expression matches the number of arrays in a jagged array initialization.
- Use collection initializers only with collection types, not with arrays or other types (**CS1925**).

Invalid collection builder

- **CS9175**: *An expression tree may not contain a collection expression.*
- **CS9177**: *The 'CollectionBuilderAttribute' builder method return type must match collection type used in 'betterness'.*
- **CS9178**: *There is no target type for the natural type 'type'.*
- **CS9179**: *Collection expression type must have an applicable constructor that can be called with no arguments.*
- **CS9180**: *The 'CollectionBuilderAttribute' builder method must be a static method.*
- **CS9181**: *The 'CollectionBuilderAttribute' builder method parameter type must match parameter 'ReadOnlySpan<{0}>'*
- **CS9182**: *Invalid 'CollectionBuilderAttribute'. No matching '{0}' method found on builder type '{1}'.*
- **CS9183**: *The 'CollectionBuilderAttribute' method return type must be a non-abstract, non-interface type usable as a 'type'*
- **CS9185**: *A static type cannot be used as a type argument.*
- **CS9186**: *The `CollectionBuilderAttribute` method name is invalid.*
- **CS9187**: *Could not find an accessible 'Create' method with the expected signature: a static method with a single parameter of type 'ReadOnlySpan<{0}>' and return type '{1}'.*
- **CS9188**: *'scoped' cannot be used as a modifier on a collection expression type parameter.*
- **CS9190**: *The 'CollectionBuilderAttribute' method 'builderMethod' is inapplicable because it's generic.*
- **CS9192**: *Inline array conversions cannot be used with collection expressions.*
- **CS9193**: *Argument 'argument' may not be passed with the 'ref' keyword.*
- **CS9194**: *Argument 'argument' may not be passed with the 'out' keyword*
- **CS9195**: *Argument 'argument' may not be passed with the 'in' keyword*
- **CS9196**: *Feature 'collection expression' is not available in C# 'version'. Please use language version 'requiredVersion' or greater.*
- **CS9197**: *Feature 'inline arrays' is not available in C# 'version'. Please use language version 'requiredVersion' or greater.*
- **CS9198**: *Feature 'ref and unsafe in async and iterator methods' is not available in C# 'version'. Please use language version 'requiredVersion' or greater.*
- **CS9199**: *Feature 'collection expression' is not available in C# 'version'. Please use language version 'requiredVersion' or greater.*
- **CS9202**: *Feature 'ref readonly parameters' is not available in C# 'version'. Please use language version 'requiredVersion' or greater.*
- **CS9208**: *'nameof' operator cannot be used on an inline array access.*
- **CS9209**: *A ref-returning property 'property' cannot be used as a value argument.*
- **CS9211**: *The expression must be of type 'type' because it's being assigned by reference*
- **CS9212**: *Cannot use collection expression as the value in a fixed statement*

- **CS9217**: *A 'ref' local cannot be preserved across 'await' or 'yield' boundary.*
- **CS9218**: *'paramName' is a ref struct and cannot be the type of a parameter*
- **CS9221**: *The type 'type' may not be a ref struct or a type parameter allowing ref structs in order to use it as parameter 'parameter' in the generic type or method 'member'*
- **CS9223**: *A struct that contains 'ref' fields cannot be used in a collection expression.*
- **CS9228**: *Non-variable declaration of a ref struct is not allowed*
- **CS9232**: *Partial method declarations have signature differences.*
- **CS9233**: *The 'file' modifier can be used only on types defined in top level in a compilation unit*

To create collection expressions with collection builder attributes correctly, follow these requirements. For more information, see Collection expressions.

- Ensure the target type has an iteration type that supports `foreach` (**CS9188**).
- Don't use generic types as collection builder types (**CS9185**).
- Verify the method name specified in `CollectionBuilderAttribute` is valid (**CS9186**).
- Apply `CollectionBuilderAttribute` only with static methods that match the required signature: return the collection type and take a `ReadOnlySpan<T>` parameter where `T` matches the element type (**CS9180**, **CS9181**, **CS9182**, **CS9183**, **CS9187**, **CS9190**).
- Ensure the return type matches and isn't abstract or an interface (**CS9177**, **CS9183**).
- Don't use ref structs or types with ref fields in collection expressions (**CS9218**, **CS9221**, **CS9223**, **CS9228**).
- Avoid using collection expressions in expression trees (**CS9175**).
- Use the correct language version for collection expressions and related features (**CS9196**, **CS9197**, **CS9198**, **CS9199**, **CS9202**).

Common language specification warnings

- **CS3007**: *Overloaded method 'method' differing only by unnamed array types is not CLS-compliant*
- **CS3016**: *Arrays as attribute arguments is not CLS-compliant*

To write CLS-compliant code with arrays, follow these guidelines. For more information, see Language independence.

- Don't create overloaded methods that differ only in array element types (**CS3007**).
- Don't use arrays as attribute arguments (**CS3016**).

CS3007 occurs if you have an overloaded method that takes a jagged array and the only difference between the method signatures is the element type of the array. To avoid this error, consider using a rectangular array rather than a jagged array or, if CLS Compliance isn't

needed, remove the CLSCompliantAttribute attribute. For more information on CLS Compliance, see Language independence and language-independent components.

CS3016 indicates that not compliant with the Common Language Specification (CLS) to pass an array to an attribute. For more information on CLS compliance, see Language independence and language-independent components.

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

---

Last updated on 11/08/2025

Resolve errors and warnings with inline array declarations

10/22/2025

This article covers the following compiler errors and warnings:

- CS9125: *Attribute parameter 'SizeConst' must be specified.*
- CS9164: *Cannot convert expression to `Span<T>` because it is not an assignable variable*
- CS9165: *Cannot convert expression to `ReadOnlySpan<T>` because it may not be passed or returned by reference*
- CS9166: *Index is outside the bounds of the inline array*
- CS9167: *Inline array length must be greater than 0.*
- CS9168: *Inline array struct must not have explicit layout.*
- CS9169: *Inline array struct must declare one and only one instance field which must not be a ref field.*
- CS9172: *Elements of an inline array type can be accessed only with a single argument implicitly convertible to `int`, `System.Index`, or `System.Range`.*
- CS9173: *An inline array access may not have a named argument specifier*
- CS9180: *Inline array element field cannot be declared as required, readonly, volatile, or as a fixed size buffer.*
- CS9181: *Inline array indexer will not be used for element access expression.*
- CS9182: *Inline array 'Slice' method will not be used for element access expression.*
- CS9183: *Inline array conversion operator will not be used for conversion from expression of the declaring type.*
- CS9184: *'Inline arrays' language feature is not supported for inline array types with element field which is either a '`ref`' field, or has type that is not valid as a type argument.*
- CS9189: *`foreach` statement on an inline array of type is not supported*
- CS9259: *Attribute System.Runtime.CompilerServices.InlineArrayAttribute cannot be applied to a record struct.*

Inline array declaration

You declare inline arrays as a `struct` type with a single field, and an attribute that specifies the length of the array. The compiler generates the following errors for invalid inline array declarations:

- **CS9125**: *Attribute parameter 'SizeConst' must be specified.*
- **CS9167**: *Inline array length must be greater than 0.*
- **CS9168**: *Inline array struct must not have explicit layout.*

- **CS9169**: *Inline array struct must declare one and only one instance field which must not be a ref field.*
- **CS9180**: *Inline array element field cannot be declared as required, readonly, volatile, or as a fixed size buffer.*
- **CS9184**: *'Inline arrays' language feature is not supported for inline array types with element field which is either a '`ref`' field, or has type that is not valid as a type argument.*
- **CS9259**: *Attribute System.Runtime.CompilerServices.InlineArrayAttribute cannot be applied to a record struct.*

To fix these arrays, ensure the following are true:

- The argument to the System.Runtime.CompilerServices.InlineArrayAttribute is a positive integer.
- The enclosing `struct` doesn't specify any explicit layout.
- The enclosing `struct` has a single instance field, and that instance field is not a `ref` field.
- The single instance field is not a fixed size buffer.
- The single instance field doesn't include the `required`, `volatile`, or `readonly` modifiers.
- Remove the `record` modifier from your inline array declaration.

Element access

You access elements of an inline array in the same way as any array. The compiler emits the following errors from incorrect element access:

- **CS9166**: *Index is outside the bounds of the inline array*
- **CS9172**: *Elements of an inline array type can be accessed only with a single argument implicitly convertible to `int`, `System.Index`, or `System.Range`.*
- **CS9173**: *An inline array access may not have a named argument specifier*
- **CS9189**: *`foreach` statement on an inline array of type is not supported*

In addition, the compiler issues the following warning when you declare an indexer:

- **CS9181**: *Inline array indexer will not be used for element access expression.*

The generated code for an inline buffer accesses the buffer memory directly, bypassing any declared indexers. Inline arrays can't be used with the `foreach` statement.

The argument to the indexer must be:

- One of these three types: `int`, a `System.Index` or a `System.Range`.
- Can't be a named argument. The compiler generates the element accessor. The parameter doesn't have a name, so you can't use named arguments.

- Is included in the bounds of the array. Like all .NET arrays, inline array element access is bounds-checked. The index must be within the bounds of the inline array.

Conversions to Span

You often use System.Span<T> or System.ReadOnlySpan<T> to work with inline arrays. The compiler generates the following errors for invalid conversions:

- **CS9164**: *Cannot convert expression to `Span<T>` because it is not an assignable variable*
- **CS9165**: *Cannot convert expression to `ReadOnlySpan<T>` because it may not be passed or returned by reference*

The compiler generates code that directly accesses the memory for an inline buffer. Therefore, some members aren't ever called. The compiler generates the following warnings if you write one of the members that aren't ever called:

- **CS9182**: *Inline array 'Slice' method will not be used for element access expression.*
- **CS9183**: *Inline array conversion operator will not be used for conversion from expression of the declaring type.*

An inline array can be implicitly converted to a `Span<T>` or `ReadOnlySpan<T>` to pass an inline array to methods. The compiler enforces restrictions on those conversions:

- The inline array must be writable in order to convert an inline array to a `Span<T>`. If the array is readonly, you can't convert it to a writable `Span<T>`. You can use `ReadOnlySpan<T>` instead.
- The *safe context* of the inline array must be at least as wide as the *safe context* of the `Span<T>` or `ReadOnlySpan<T>` for the conversion to succeed. You must either limit the context of the span, or expand the scope of the inline array.

In addition, the compiler never generates calls to a `Slice` method in an inline buffer. Conversion operators to convert an inline buffer to a `Span` or `ReadOnlySpan` aren't called. The compiler generates code to create a System.Span<T> or System.ReadOnlySpan<T> directly from the memory buffer.

Errors and warnings when using lambda expressions and anonymous functions

10/22/2025

There are several *errors* related to declaring and using lambda expressions:

- CS0748: *Inconsistent lambda parameter usage; parameter types must be all explicit or all implicit.*
- CS1621: *The yield statement cannot be used inside an anonymous method or lambda expression.*
- CS1628: *Cannot use* `in` `ref` *or* `out` *parameter inside an anonymous method, lambda expression, or query expression.*
- CS1632: *Control cannot leave the body of an anonymous method or lambda expression.*
- CS1673: *Anonymous methods, lambda expressions, and query expressions inside structs cannot access instance members of 'this'.*
- CS1686: *Local variable or its members cannot have their address taken and be used inside an anonymous method or lambda expression.*
- CS1706: *Expression cannot contain anonymous methods or lambda expressions.*
- CS8030: *Anonymous function converted to a void returning delegate cannot return a value.*
- CS8175: *Cannot use ref local inside an anonymous method, lambda expression, or query expression.*
- CS8916: *Attributes on lambda expressions require a parenthesized parameter list.*
- CS8971: *InterpolatedStringHandlerArgument has no effect when applied to lambda parameters and will be ignored at the call site.*
- CS8975: *The contextual keyword* `var` *cannot be used as an explicit lambda return type.*
- CS9098: *Implicitly typed lambda parameter '...' cannot have a default value.*

In addition, there are several *warnings* related to declaring and using lambda expressions:

- CS8971: *InterpolatedStringHandlerArgument has no effect when applied to lambda parameters and will be ignored at the call site.*
- CS9099: *The default parameter value does not match in the target delegate type.*
- CS9100: *Parameter has params modifier in lambda but not in target delegate type.*

The compiler also produces the following *informational* message:

- CS9236: *Compiling requires binding the lambda expression at least count times. Consider declaring the lambda expression with explicit parameter types, or if the containing method call is generic, consider using explicit type arguments.*

Syntax limitations in lambda expressions

Some C# syntax is prohibited in lambda expressions and anonymous methods. Using invalid constructs in a lambda expression causes the following errors:

- **CS1621**: *The `yield` statement cannot be used inside an anonymous method or lambda expression.*
- **CS1628**: *Cannot use `in`, `ref`, or `out` parameter inside an anonymous method, lambda expression, or query expression.*
- **CS1632**: *Control cannot leave the body of an anonymous method or lambda expression.*
- **CS1673**: *Anonymous methods, lambda expressions, and query expressions inside structs cannot access instance members of `this`.*
- **CS1686**: *Local variable or its members cannot have their address taken and be used inside an anonymous method or lambda expression.*
- **CS8175**: *Cannot use ref local inside an anonymous method, lambda expression, or query expression.*

All the following constructs are disallowed in lambda expressions:

- `yield` statements (`yield return` or `yield break`)
- Calling a method that has an `in`, `ref`, or `out` parameter
- `ref` local variables
- `break`, `goto`, and `continue` statements
- `this` access when `this` is a `struct` type
- Anonymous methods or lambda expressions inside another expression, such as an Attribute constructor.

You can't use any of these constructs in a lambda expression or an anonymous method. Many are allowed in a local function.

In addition, interpolated string handler types are ignored when applied to a lambda parameter. If you use one, you see the following warning:

- **CS8971**: *InterpolatedStringHandlerArgument has no effect when applied to lambda parameters and will be ignored at the call site.*

Certain expressions cause the compiler to emit the following informational warning:

- **CS9236**: *Compiling requires binding the lambda expression at least count times. Consider declaring the lambda expression with explicit parameter types, or if the containing method call is generic, consider using explicit type arguments.*

The complexity of the lambda expressions and how they invoke other lambda expressions is negatively impacting compiler performance. The reason is that the compiler must infer parameter and argument types through the lambda expressions and the potential types takes time.

Lambda expression parameters and returns

These errors indicate a problem with a parameter declaration:

- **CS0748**: *Inconsistent lambda parameter usage; parameter types must be all explicit or all implicit.*
- **CS9098**: *Implicitly typed lambda parameter '...' cannot have a default value.*
- **CS8030**: *Anonymous function converted to a void returning delegate cannot return a value.*
- **CS8916**: *Attributes on lambda expressions require a parenthesized parameter list.*
- **CS8975**: *The contextual keyword 'var' cannot be used as an explicit lambda return type.*

Lambda expression parameters must follow these rules:

- When a lambda expression has multiple parameters, either all parameters must be explicitly typed or all parameters must be implicitly typed.
- All lambda parameters with a default value must be explicitly typed.
- If attributes are applied to any parameters, the parameter list must be enclosed in parentheses.

Return types of lambda expression must follow these rules:

- A lambda expression that returns any value can't be converted to a `void` returning delegate, such as `Action`.
- The return type is either inferred, or an explicit type. A return type declared using the keyword `var` isn't allowed.

Lambda expression delegate type

- CS9099: Warning: *The default parameter value does not match in the target delegate type.*

When you declare a default value or add the `params` modifier with a lambda expression parameter, the delegate type isn't one of the `Func` or `Action` types. Rather, it's a custom type that includes the default parameter value or `params` modifier. The following code generates warnings because it assigns a lambda expression that has a default parameter to an `Action` type:

```
C#
```

```
Action<int> a1 = (int i = 2) => { };
Action<string[]> a3 = (params string[] s) => { };
```

To fix the error, either remove the default parameter or use an implicitly typed variable for the delegate type:

C#

```
Action<int> a1 = (int i) => { };
var a2 = (int i = 2) => { };
Action<string[]> a3 = (string[] s) => { };
var a4 = (params string[] s) => { };
```

Resolve errors and warnings that affect overload resolution.

Article • 09/14/2024

This article covers the following compiler errors:

- CS0034 - *Operator is ambiguous on operands of type '`type1`' and '`type2`'*
- CS0035 - *Operator is ambiguous on an operand of type 'type'*
- CS0111 - *Type already defines a member called 'name' with the same parameter types*
- CS0121 - *The call is ambiguous between the following methods or properties*
- CS0457 - *Ambiguous user defined conversions when converting from 'type1' to 'type2'*
- CS0663 - *Cannot define overloaded methods that differ only on `ref` and `out`.*
- CS1007 - *Property accessor already defined*
- CS1019 - *Overloadable unary operator expected*
- CS1020 - *Overloadable binary operator expected*
- CS1501 - *No overload for method 'method' takes 'number' arguments*
- CS1534 - *Overloaded binary operator 'operator' takes two parameters*
- CS1535 - *Overloaded unary operator 'operator' takes one parameter*
- CS1928 - *'Type' does not contain a definition for 'method' and the best extension method overload 'method' has some invalid arguments.*
- CS1929 - *'TypeA' does not contain a definition for 'method' and the best extension method overload 'TypeB.method' requires a receiver of type 'TypeC'*
- CS8315 - *Operator is ambiguous on operands*
- CS9261 - *Cannot use '`OverloadResolutionPriorityAttribute`' on an overriding member.*
- CS9262 - *Cannot use '`OverloadResolutionPriorityAttribute`' on this member.*

In addition, the following compiler warning:

- CS3006 - *Overloaded method 'method' differing only in `ref` or `out`, or in array rank, is not CLS-compliant*

Incorrect overload definition

- **CS0663** - *Cannot define overloaded methods that differ only on `ref` and `out`.*
- **CS1019** - *Overloadable unary operator expected*
- **CS1020** - *Overloadable binary operator expected*

- **CS1534** - *Overloaded binary operator 'operator' takes two parameters*
- **CS1535** - *Overloaded unary operator 'operator' takes one parameter*

In addition, the following compiler warning:

- **CS3006** - *Overloaded method 'method' differing only in `ref` or `out`, or in array rank, is not CLS-compliant*

When you create overloaded operators in your class, the signature must match the number of parameters required for that operator. You have the wrong number of parameters in the operator definition.

In addition, overload operators must use the defined operator name. The only exception is when you create a conversion operator, where the operator name matches the return type of the conversion.

Duplicate overloads defined

- **CS0111** - *Type already defines a member called 'name' with the same parameter types*
- **CS1007** - *Property accessor already defined*

These errors indicate that you created overloads that aren't different. Common reasons include:

- The matching parameter types have an identity conversion. For example `M(dynamic)` and `M(object)`, or `M2(string)` and `M2(string?)`.
- The matching parameter types differ only in reference modifier. For example, `M(ref int)` and `M(out int)`.
- You've included both an `init` and a `set` accessor for a property.

No overload found

- **CS1501** - *No overload for method 'method' takes 'number' arguments*
- **CS1928** - *'Type' does not contain a definition for 'method' and the best extension method overload 'method' has some invalid arguments.*
- **CS1929** - *'TypeA' does not contain a definition for 'method' and the best extension method overload 'TypeB.method' requires a receiver of type 'TypeC'*

Your code calls a method where the name exists, but some arguments are incorrect, or you've used the wrong number of arguments. If you're trying to call an extension method, the containing class might be in a namespace that's not imported.

> **Note**
>
> Error `CS1928` isn't used by the latest compilers. Newer compilers use `CS1929` exclusively.

Ambiguous overloads

- **CS0034** - *Operator is ambiguous on operands of type '`type1`' and '`type2`'*
- **CS0035** - *Operator is ambiguous on an operand of type 'type'*
- **CS0121** - *The call is ambiguous between the following methods or properties*
- **CS0457** - *Ambiguous user defined conversions when converting from 'type1' to 'type2'*
- **CS8315** - *Operator is ambiguous on operands*

These errors indicate there isn't one better overload than others. There are several possible causes for these errors:

- Different arguments are better for each parameter. One might be an exact match and the other requires a conversion:

  ```
  C#

  void M(int a, double b) {    }
  void M(double a, int b) {}

  M(1, 1); // ambiguous
  ```

- Multiple user defined conversion operators are candidates. Conversion operators can be declared in the source type, its base types, the destination type, or its base types. If a conversion operator was defined in more than one, the compiler can't pick.

- Multiple conversion operators could perform a compatible conversion.

- One or more arguments uses `default` (not `default(T)`) and multiple types could be chosen.

In most of these cases, adding an explicit cast can specify which overload should be chosen.

Overload resolution priority

- **CS9261** - *Cannot use '`OverloadResolutionPriorityAttribute`' on an overriding member.*
- **CS9262** - *Cannot use '`OverloadResolutionPriorityAttribute`' on this member.*

Your code violated the rules for using the OverloadResolutionPriorityAttribute to favor one overload instead of another. You can't apply the `OverloadResolutionPriorityAttribute` to the following method types:

- Non-indexer properties
- Property, indexer, or event accessors
- Conversion operators
- Lambdas
- Local functions
- Finalizers
- Static constructors

In addition, you can't apply the `OverloadResolutionPriorityAttribute` to an `override` of a `virtual` or `abstract` member. The compiler uses the value from the base type declaration.

Resolve errors and warnings generated from expressions prohibited in expression trees

05/27/2025

This article covers the following compiler errors:

- **CS0765** - *Partial methods with only a defining declaration or removed conditional methods cannot be used in expression trees.*
- **CS0831** - *An expression tree may not contain a base access.*
- **CS0832** - *An expression tree may not contain an assignment operator*
- **CS0834** - *A lambda expression with a statement body cannot be converted to an expression tree.*
- **CS0835** - *Cannot convert lambda to an expression tree whose type argument 'type' is not a delegate type.*
- **CS0838** - *An expression tree may not contain a multidimensional array initializer.*
- **CS0845** - *An expression tree lambda may not contain a coalescing operator with a null or default literal left-hand side.*
- **CS0853** - *An expression tree may not contain a named argument specification.*
- **CS0854** - *An expression tree may not contain a call or invocation that uses optional arguments.*
- **CS0855** - *An expression tree may not contain an indexed property.*
- **CS1944** - *An expression tree may not contain an unsafe pointer operation.*
- **CS1945** - *An expression tree may not contain an anonymous method expression.*
- **CS1946** - *An anonymous method expression cannot be converted to an expression tree.*
- **CS1951** - *An expression tree lambda may not contain a `ref`, `in` or `out` parameter.*
- **CS1952** - *An expression tree lambda may not contain a method with variable arguments.*
- **CS1963** - *An expression tree may not contain a dynamic operation.*
- **CS1989** - *Async lambda expressions cannot be converted to expression trees.*
- **CS2037** - *An expression tree lambda may not contain a COM call with ref omitted on arguments.*
- **CS7053** - *An expression tree may not contain "feature".*
- **CS8072** - *An expression tree lambda may not contain a null propagating operator.*
- **CS8074** - *An expression tree lambda may not contain a dictionary initializer.*
- **CS8075** - *An extension `Add` method is not supported for a collection initializer in an expression lambda.*
- **CS8110** - *An expression tree may not contain a reference to a local function.*
- **CS8122** - *An expression tree may not contain an '`is`' pattern-matching operator.*
- **CS8143** - *An expression tree may not contain a tuple literal.*

- **CS8144** - *An expression tree may not contain a tuple conversion.*
- **CS8153** - *An expression tree lambda may not contain a call to a method, property, or indexer that returns by reference.*
- **CS8155** - *Lambda expressions that return by reference cannot be converted to expression trees.*
- **CS8188** - *An expression tree may not contain a throw-expression.*
- **CS8198** - *An expression tree may not contain an out argument variable declaration.*
- **CS8207** - *An expression tree may not contain a discard.*
- **CS8382** - *An expression tree may not contain a tuple `==` or `!=` operator.*
- **CS8514** - *An expression tree may not contain a switch expression.*
- **CS8640** - *Expression tree cannot contain value of ref struct or restricted type.*
- **CS8642** - *An expression tree may not contain a null coalescing assignment.*
- **CS8790** - *An expression tree may not contain a pattern System.Index or System.Range indexer access.*
- **CS8791** - *An expression tree may not contain a from-end index ('`^`') expression.*
- **CS8792** - *An expression tree may not contain a range ('`..`') expression.*
- **CS8810** - '`&`' *on method groups cannot be used in expression trees.*
- **CS8849** - *An expression tree may not contain a `with`-expression.*
- **CS8927** - *An expression tree may not contain an access of static virtual or abstract interface member.*
- **CS8952** - *An expression tree may not contain an interpolated string handler conversion.*
- **CS8972** - *A lambda expression with attributes cannot be converted to an expression tree.*
- **CS9170** - *An expression tree may not contain an inline array access or conversion.*
- **CS9175** - *An expression tree may not contain a collection expression.*
- **CS9226** - *An expression tree may not contain an expanded form of non-array params collection parameter.*
- **CS9296** - *An expression tree may not contain an extension property access.*
- **CS9307** - *An expression tree may not contain a named argument specification out of position.*

Expression tree restrictions

All of the errors in the preceding list indicate you've used a C# expression type that isn't allowed in an expression tree. In most cases, the prohibited expressions represent syntax introduced after C# 3.0. These expressions are prohibited because allowing them would create a breaking change in all libraries that parse expression trees. All libraries would need to be enhanced to parse new C# expressions if newer constructs were allowed.

The following expressions are prohibited:

- Invocations of partial methods that don't have an implementing declaration.

- Invocations of conditional methods that have been removed.
- Invocations of local functions.
- `async` lambda expressions aren't allowed.
- Using base access to directly call a virtual method declared in a base class.
- assignment operations.
- statement lambdas aren't allowed.
- multi-dimensional array initializers. Instead, you must create and initialize a multi-dimensional array outside of the expression tree.
- dynamic operations aren't allowed.
- pattern matching expressions aren't allowed.
- Tuple literals and many tuple operations, such as equality comparisons aren't allowed.
- throw expressions aren't allowed.
- discard (`_`) declarations.
- The index and range operators aren't allowed.
- Non-destructive mutation using with expressions aren't allowed.
- You can't declare or access inline arrays.
- You can't include collection expressions.
- The null propagating and null coalescing operators aren't allowed.
- ref struct types, such as System.Span<T> and System.ReadOnlySpan<T> aren't allowed.
- `in`, `out`, and `ref` parameters, including `out` variable declarations, aren't allowed.
- `ref` returns aren't allowed.
- Calls to methods that return by `ref` aren't allowed.
- static abstract interface members can't be accessed.
- Inline arrays.
- The `params` modifier is allowed only on single-dimensional arrays. Other collection types aren't allowed.

Other restrictions are:

- Extension properties can't be accessed as extensions.
- Attributes can't be applied to the lambda expression, its parameters or return.
- The lambda expression must be convertible to a type derived from System.Linq.Expressions.Expression whose type parameter is a delegate type.
- named and optional parameters are restricted. The expression can't call a method specifying named arguments, and it can't use the default value of an optional parameter.
- Dictionary initializers aren't allowed. Neither are extension `Add` methods.
- The target expression must be a lambda expression. Constants and variables aren't allowed, but a lambda expression that returns a constant or variable is.
- Unsafe pointer operations aren't allowed.
- COM calls must include `ref` on arguments; it can't be implied.
- The unsupported `__arglist` keyword isn't allowed.

Resolve warnings related using namespaces

Article • 11/03/2023

This article covers the following compiler errors:

- CS0138: *Error: A using namespace directive can only be applied to namespaces; 'type' is a type not a namespace.*
- CS0431: *Error: Cannot use alias 'identifier' with `::` since the alias references a type. Use `.` instead.*
- CS0432: *Error: Alias 'identifier' not found.*
- CS0576: *Error: Namespace 'namespace' contains a definition conflicting with alias 'identifier'.*
- CS0687: *Error: The namespace alias qualifier `::` always resolves to a type or namespace so is illegal here. Consider using `.` instead.*
- CS1529: *Error: A using clause must precede all other elements defined in the namespace except extern alias declarations.*
- CS1537: *Error: The using alias 'alias' appeared previously in this namespace.*
- CS7000: *Error: Unexpected use of an aliased name.*
- CS7007: *Error: A `using static` directive can only be applied to types. Consider a `using namespace` directive instead*
- CS8083: *Error: An alias-qualified name is not an expression.*
- CS8085: *Error: A 'using static' directive cannot be used to declare an alias.*
- CS8914: *Error: A global using directive cannot be used in a namespace declaration.*
- CS8915: *Error: A global using directive must precede all non-global using directives.*
- CS9055: *Error: A file-local type cannot be used in a 'global using static' directive.*
- CS9130: *Error: Using alias cannot be a `ref` type.*
- CS9131: *Error: Only a using alias can be `unsafe`.*
- CS9132: *Error: Using alias cannot be a nullable reference type.*
- CS9133: *Error: `static` modifier must precede `unsafe` modifier.*
- CS9162: *Type is not valid for 'using static'. Only a class, struct, interface, enum, delegate, or namespace can be used.*

And the following compiler warnings:

- CS0105: *Warning: The using directive for 'namespace' appeared previously in this namespace.*
- CS0440: *Warning: Defining an alias named `global` is ill-advised since `global::` always references the global namespace and not an alias.*

- CS8019: *Info: Unnecessary using directive.*
- CS8933: *Info: The using directive appeared previously as global using.*

These errors and warnings indicate you're `using` directive isn't formed correctly. The following sections cover these errors and how to correct them.

Using directive

The `using` directive must precede any other elements in a `namespace` declaration, or before any `namespace` declarations in the file. Putting a `using` directive later in the file causes the compiler to produce error **CS1529**:

```
C#

namespace UsingDirective;
public class MyClass
{
}

using System.Text.Json; // CS1529
```

To fix this issue, move any `using` declarations to the top of the file or the top of the namespace:

```
C#

using System.Text.Json;
namespace UsingDirective;
public class MyClass
{
}
```

The compiler produces warning **CS8933**, **CS0105** or diagnostic **CS8019** for a duplicate `using` directive from a `using` or `global using` directive. You can remove any duplicates.

Incorrectly combining a `using` directive with the `static`, `global`, or `unsafe` modifiers on a `using` directive are covered later in this article.

Using static directive

The `using static` directive imports one type's members into the current namespace. The following example imports the methods from `System.Console`, such as `WriteLine` into the current namespace:

```
C#

using static System.Console;
```

The compiler generates **CS0138** if you omit the `static` modifier:

```
C#

using System.Console; // CS0138
```

The compiler generates **CS7007** if you include the `static` modifier importing namespace instead of a type:

```
C#

using static System; // CS7007
```

The compiler emits CS9162 if the symbol isn't one of the proper types.

If you combine the `static` modifier with the `unsafe` modifier in a `using` directive, the `static` modifier must come first:

```
C#

using static unsafe UnsafeExamples.UnsafeType;
```

Global using directive

A `global using` directive imports the namespace or type in all source files in the current project:

```
C#

global using System.Text;
```

Any `global using` directives must precede any non-global `using` directives in that source file, and must not be placed in a `namespace`. Doing so results in **CS8915** and **CS8914**, respectively.

Furthermore, a `static global using` directive can't reference a file-local type.

Alias qualifier

The alias qualifier, ::, precedes a namespace alias, or follows the `global` alias. If you use `::` where `.` should be used to separate elements of a fully qualified name, the compiler emits one of **CS0431**, **CS0432**, **CS0687**, **CS7000*, or **CS8083**.

In all cases, replace the `::` with the `.` separator.

In addition, if you define an alias named `global`, the compiler issues **CS0440**. The `global` alias always refers to the global namespace. Declaring an alias for it doesn't work, and you should pick a different name for your alias.

Alias name conflicts

You can declare an alias to a namespace or a type with a `using` directive:

```
C#
```

```
using JSON = System.Text.Json;
using ValueMap = System.Collections.Generic.Dictionary<string, decimal>;
using TimedData = (System.DateTime timeRecorded, decimal value);
```

You should try to create a unique name for the alias, the name on the left of the `=` sign in the preceding examples. Using a name that already maps to a Type (for example `Object`) or a namespace (`System`) can cause **CS0576** or **CS1537**.

Restrictions on using aliases

Prior to C# 12, the language imposed these restrictions on `using` directives that create an alias for a type declaration:

- You can't create an alias with a `using static` directive:

  ```
  C#
  ```

  ```
  using static con = System.Console;
  using static unsafe ip = int*;
  ```

Beginning with C# 12, these restrictions are introduced:

- You can't use the `in`, `ref`, or `out` modifiers in a using alias:

C#

```
// All these are invalid
using RefInt = ref int;
using OutInt = out int;
using InInt = in int;
```

- An `unsafe using` directive must specify an alias, or a `static using`:

C#

```
// Elsewhere:
public namespace UnsafeExamples
{
    public unsafe static class UnsafeType
    {
        // ...
    }
}

// Using directives:
using unsafe IntPointer = int*;
using static unsafe UnsafeExamples.UnsafeType;
using unsafe UnsafeExamples; // not allowed
```

- You can't create an alias to a nullable reference type:

C#

```
using NullableInt = System.Int32?; // Allowed
using NullableString = System.String?; // Not allowed
```

Resolve warnings related to the `using` statements and `using` declarations

10/28/2025

This article covers the following compiler errors:

- CS0245: *Destructors and `object.Finalize` cannot be called directly. Consider calling `IDisposable.Dispose` if available.*
- CS0728: *Possibly incorrect assignment to local variable which is the argument to a using or `lock` statement. The `Dispose` call or unlocking will happen on the original value of the local.*
- CS1674: *Type used in a using statement must be implicitly convertible to '`System.IDisposable`'.*
- CS8410: *'Type used in an asynchronous `using` statement must be implicitly convertible to '`System.IAsyncDisposable`' or implement a suitable '`DisposeAsync`' method.*
- CS8417: *Type used in an asynchronous using statement must implement '`System.IAsyncDisposable`' or implement a suitable '`DisposeAsync`' method. Did you mean '`using`' rather than '`await using`'?*
- CS8418: *Type used in a using statement must implement '`System.IDisposable`'. Did you mean '`await using`' rather than 'using'?*
- CS8647: *A using variable cannot be used directly within a switch section (consider using braces).*
- CS8648: *A `goto` cannot jump to a location after a using declaration.*
- CS8649: *A `goto` cannot jump to a location before a using declaration within the same block.*
- CS9229: *Modifiers cannot be placed on using declarations.*

Implementing IDisposable and IAsyncDisposable

The following compiler errors and warnings indicate issues with implementing or using the dispose pattern:

- **CS0245**: *Destructors and `object.Finalize` cannot be called directly. Consider calling `IDisposable.Dispose` if available.*
- **CS1674**: *Type used in a using statement must be implicitly convertible to '`System.IDisposable`'.*
- **CS8410**: *Type used in an asynchronous `using` statement must be implicitly convertible to '`System.IAsyncDisposable`' or implement a suitable 'DisposeAsync' method.*

- **CS8417**: *Type used in an asynchronous `using` statement must implement '`System.IAsyncDisposable`' or implement a suitable '`DisposeAsync`' method. Did you mean '`using`' rather than '`await using`'?*
- **CS8418**: *Type used in a using statement must implement '`System.IDisposable`'. Did you mean '`await using`' rather than '`using`'?*

The using statement ensures proper disposal of resources at the end of the `using` block. To use a type with a `using` statement, it must implement the appropriate disposal interface. For synchronous `using` statements, the type must implement IDisposable. For asynchronous `await using` statements, the type must implement IAsyncDisposable.

- **Cannot call Finalize directly (CS0245)**: You can't directly call a destructor or the Object.Finalize method. The garbage collector automatically invokes finalizers when objects are no longer referenced. For deterministic cleanup, implement IDisposable and call the `Dispose` method instead.
- **Type must implement IDisposable (CS1674)**: Only types that implement IDisposable can be used in a `using` statement. Value types don't implement this interface, and generic type parameters without proper constraints can't be assumed to be disposable. Apply a type constraint like `where T : IDisposable` when working with generic types.
- **Type must implement IAsyncDisposable (CS8410)**: Types used with `await using` must implement IAsyncDisposable or provide a suitable `DisposeAsync` method. If your type doesn't support asynchronous disposal, use a synchronous `using` statement instead or implement the required interface.
- **Mismatched disposal pattern (CS8417, CS8418)**: CS8417 occurs when you use `await using` with a type that only implements IDisposable. CS8418 occurs when you use synchronous `using` with a type that only implements IAsyncDisposable. Match the `using` keyword to the interface your type implements, or implement both interfaces if you need to support both patterns.

For more information, see Finalizers, Implement a Dispose method, and Implement a DisposeAsync method.

Using variable scope and control flow

The following compiler errors and warnings relate to incorrect usage of `using` variables within control flow statements:

- **CS0728**: *Possibly incorrect assignment to local variable which is the argument to a `using` or `lock` statement. The `Dispose` call or unlocking will happen on the original value of the local.*

- **CS8647**: *A using variable cannot be used directly within a switch section (consider using braces).*
- **CS8648**: *A `goto` cannot jump to a location after a using declaration.*
- **CS8649**: *A `goto` cannot jump to a location before a using declaration within the same block.*

Variables declared with `using` have specific scoping rules that prevent resource leaks. The compiler enforces these rules to ensure proper disposal.

- **Assignment to using variable (CS0728)**: This warning indicates you assigned a new value to a variable that's the resource in a `using` statement. The dispose call occurs on the original value, not the newly assigned value, which can lead to resource leaks. Initialize the resource in the `using` statement declaration instead of assigning to it later.

- **Using variable in switch section (CS8647)**: A `using` declaration creates a variable that's disposed at the end of its scope. When used directly in a switch section without braces, the scope is ambiguous and can lead to errors. Wrap the switch section content in braces to clearly define the scope.

- **Goto statements and using declarations (CS8648, CS8649)**: You can't use `goto` statements to jump over `using` declarations because jumping would skip proper resource management. CS8648 occurs when jumping forward over a `using` declaration, and CS8649 occurs when jumping backward to a location before a `using` declaration. Restructure your code to use structured control flow like loops, or move the `using` declaration outside the jump target.

For more information, see using statement.

Incorrect `using` declaration

- **CS9229**: *Modifiers cannot be placed on using declarations.*

A variable declaration wrapped in a `using` declaration can't include any of the following modifiers:

- `const`
- `static`
- `volatile`
- `readonly`
- Accessibility modifiers: `public`, `protected`, `internal`, `private`, `protected internal`, or `private protected`

The following example generates CS9229:

```C#
using System;

class Program
{
    static void Main()
    {
        // error CS9229: Modifiers cannot be placed on using declarations.
        public using var resource = new Resource();

        // error CS9229: Modifiers cannot be placed on using declarations.
        static using var anotherResource = new Resource();
    }
}

class Resource : IDisposable
{
    public void Dispose() { }
}
```

To correct this error, remove the modifier from the `using` declaration:

```C#
using var resource = new Resource();
```

For more information, see using statement.

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

Errors and warnings associated with source generators and interceptors

10/22/2025

The following errors are generated when source generators or interceptors are loaded during a compilation:

- CS9137: *The 'interceptors' experimental feature is not enabled. Add* `<Features>InterceptorsPreview<Features>` *to your project.*
- CS9138: *Method cannot be used as an interceptor because it or its containing type has type parameters.*
- CS9139: *Cannot intercept: compilation does not contain a file with path.*
- CS9140: *Cannot intercept: compilation does not contain a file with path. Did you mean to use a different path?*
- CS9141: *The provided line and character number does not refer to an interceptable method name, but rather to a token.*
- CS9142: *The given file has* `n` *lines, which is fewer than the provided line number* `m`*.*
- CS9143: *The given line is* `c` *characters long, which is fewer than the provided character number* `n`*.*
- CS9144: *Cannot intercept method* `M` *with interceptor* `V` *because the signatures do not match.*
- CS9145: *Cannot intercept: Path is unmapped. Expected mapped path.*
- CS9146: *An interceptor method must be an ordinary member method.*
- CS9147: *The provided line and character number does not refer to the start of a token. Did you mean to use line* `n` *and character* `c`*?*
- CS9148: *Interceptor must have a* `this` *parameter matching parameter.*
- CS9149: *Interceptor must not have a* `this` *parameter because method does not have a* `this` *parameter.*
- CS9150: *Interceptor cannot have a* `null` *file path.*
- CS9151: *Possible method name* `M` *cannot be intercepted because it is not being invoked.*
- CS9152: *Cannot intercept a call in file with this path because multiple files in the compilation have this path.*
- CS9153: *The indicated call is intercepted multiple times.*
- CS9155: *Cannot intercept call with* `M` *because it is not accessible within* `V`*.*
- CS9156: *Cannot intercept call to* `M` *with* `V` *because of a difference in 'scoped' modifiers or* `[UnscopedRef]` *attributes.*
- CS9157: *Line and character numbers provided to* `InterceptsLocationAttribute` *must be positive.*

- CS9160: *A nameof operator cannot be intercepted.*
- CS9161: *An interceptor cannot be marked with* `UnmanagedCallersOnlyAttribute`.
- CS9177: *Interceptor must be non-generic or have matching arity.*
- CS9178: *Method must be non-generic to match*
- CS9206: *An interceptor cannot be declared in the global namespace.*
- CS9207: *Cannot intercept because method is not an invocation of an ordinary member method.*
- CS9231: *The data argument to InterceptsLocationAttribute is not in the correct format.*
- CS9232: *Version 'version' of the interceptors format is not supported. The latest supported version is '1'.*
- CS9233: *Cannot intercept a call in file 'file' because it is duplicated elsewhere in the compilation.*
- CS9234: *Cannot intercept a call in file 'file' because a matching file was not found in the compilation.*
- CS9235: *The data argument to InterceptsLocationAttribute refers to an invalid position in file 'file'.*

The following warnings are generated when source generators or interceptors are loaded during a compilation:

- CS8784: *Generator '*`YourSourceGeneratorName`*' failed to initialize. It will not contribute to the output and compilation errors may occur as a result.*
- CS8785: *Generator '*`YourSourceGeneratorName`*' failed to generate source. It will not contribute to the output and compilation errors may occur as a result.*
- CS9154: *Intercepting a call to* `M` *with interceptor* `V`*, but the signatures do not match.*
- CS9158: *Nullability of reference types in return type doesn't match interceptable method.*
- CS9159: *Nullability of reference types in type of parameter doesn't match interceptable method.*
- CS9270: *'*`InterceptsLocationAttribute(string, int, int)`*' is not supported. Move to 'InterceptableLocation'-based generation of these attributes instead. (https://github.com/dotnet/roslyn/issues/72133)*

These errors and warnings follow these themes:

Interceptors are experimental

This error indicates you must enable the experimental feature.

- **CS9137**: *The 'interceptors' experimental feature is not enabled. Add* `<Features>InterceptorsPreview<Features>` *to your project.*

In C# 12, interceptors are experimental. Interceptors are 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, you must set the `<Features>InterceptorsPreview<Features>` element in your project file. Without this flag, interceptors are disabled, even when other C# 12 features are enabled.

Signature mismatch

The following errors indicate a mismatch between the interceptor method and the interceptable method, or a violation of the rules regarding interceptor method declarations:

- **CS9144**: *Cannot intercept method `M` with interceptor `V` because the signatures do not match.*
- **CS9148**: *Interceptor must have a `this` parameter matching parameter.*
- **CS9149**: *Interceptor must not have a `this` parameter because method does not have a `this` parameter.*
- **CS9155**: *Cannot intercept call with `M` because it is not accessible within `V`.*
- **CS9156**: *Cannot intercept call to `M` with `V` because of a difference in 'scoped' modifiers or `[UnscopedRef]` attributes.*
- **CS9177**]: *Interceptor must be non-generic or have matching arity.*
- **CS9178**: *Method must be non-generic to match*

In addition, the following warnings indicate a mismatch in the signatures of the interceptor and the interceptable method:

- **CS9154**: *Intercepting a call to `M` with interceptor `V`, but the signatures do not match.*
- **CS9158**: *Nullability of reference types in return type doesn't match interceptable method.*
- **CS9159**: *Nullability of reference types in type of parameter doesn't match interceptable method.*
- **CS9270**: '`InterceptsLocationAttribute(string, int, int)`' *is not supported. Move to 'InterceptableLocation'-based generation of these attributes instead. (https://github.com/dotnet/roslyn/issues/72133)*

The interceptor method must be compatible with the interceptable method. You must follow these rules:

- Instance methods can intercept instance methods, not static methods. Similarly, static methods can only intercept static methods, not instance methods.
- The method signatures for the interceptor and the interceptable method must match: They must have the same parameters with the same modifiers in the same order. The

return types must also match.
- The ref safe contexts must match. In other words, corresponding `ref` parameters must be either `scoped` or not `scoped`.
- They methods must both be non-generic, or both must have the same number of type parameters.
- An updated `InterceptorLocationAttribute` constructor signature is preferred.

Incorrect mapping

Interceptors require a source mapping that maps the interceptable method and the interceptor method. The following errors indicate an issue with the mapping:

- **CS9139**: *Cannot intercept: compilation does not contain a file with path.*
- **CS9140**: *Cannot intercept: compilation does not contain a file with path. Did you mean to use a different path?*
- **CS9141**: *The provided line and character number does not refer to an interceptable method name, but rather to a token.*
- **CS9142**: *The given file has `n` lines, which is fewer than the provided line number `m`.*
- **CS9143**: *The given line is `c` characters long, which is fewer than the provided character number `n`.*
- **CS9145**: *Cannot intercept: Path is unmapped. Expected mapped path.*
- **CS9147**: *The provided line and character number does not refer to the start of a token. Did you mean to use line `n` and character `c`?*
- **CS9150**: *Interceptor cannot have a `null` file path.*
- **CS9157**: *Line and character numbers provided to `InterceptsLocationAttribute` must be positive.*

Incorrect interceptor declaration

The following errors indicate issues with interceptor declarations, including problems with the `InterceptsLocationAttribute` format or violations of interceptor rules:

- **CS9231**: *The data argument to InterceptsLocationAttribute is not in the correct format.*
- **CS9232**: *Version 'version' of the interceptors format is not supported. The latest supported version is '1'.*
- **CS9233**: *Cannot intercept a call in file 'file' because it is duplicated elsewhere in the compilation.*
- **CS9234**: *Cannot intercept a call in file 'file' because a matching file was not found in the compilation.*

- **CS9235**: *The data argument to InterceptsLocationAttribute refers to an invalid position in file 'file'.*
- **CS9138**: *Method cannot be used as an interceptor because it or its containing type has type parameters.*
- **CS9146**: *An interceptor method must be an ordinary member method.*
- **CS9151**: *Possible method name `M` cannot be intercepted because it is not being invoked.*
- **CS9152**: *Cannot intercept a call in file with this path because multiple files in the compilation have this path.*
- **CS9153**: *The indicated call is intercepted multiple times.*
- **CS9160**: *A nameof operator cannot be intercepted.*
- **CS9161**: *An interceptor cannot be marked with `UnmanagedCallersOnlyAttribute`.*
- **CS9206**: *An interceptor cannot be declared in the global namespace.*
- **CS9207**: *Cannot intercept because method is not an invocation of an ordinary member method.*

These errors occur when interceptor declarations violate the rules for interceptors or when the `InterceptsLocationAttribute` contains invalid data:

- **CS9231** is generated when the data format doesn't match the expected structure. The attribute requires specifically formatted data that encodes the file path and position information.
- **CS9232** indicates you're using a version number that isn't supported. The interceptors feature uses version '1' format. Update your source generator to use the supported version.
- **CS9233** happens when the compilation contains multiple files with the same path, making it ambiguous which file to intercept in. Ensure each file in your compilation has a unique path.
- **CS9234** is emitted when the file path specified in the attribute doesn't match any file in the current compilation. Verify the file path is correct and the file is included in the compilation.
- **CS9235** occurs when the position data (line and character numbers) points to an invalid location in the specified file. This can happen if the position is outside the file's bounds or doesn't correspond to a valid interception point.
- **CS9138** prevents generic methods or members of generic classes from being interceptors.
- **CS9146** requires interceptors to be ordinary members (not operators, constructors, or finalizers).
- **CS9151** prevents intercepting methods that are never invoked.
- **CS9152** detects duplicate file paths in the compilation.
- **CS9153** prevents multiple interceptions of the same call.
- **CS9160** prevents intercepting `nameof` operators.
- **CS9161** prevents interceptors from being marked with `UnmanagedCallersOnlyAttribute`.

- **CS9206** requires interceptors to be contained in a namespace (not global).
- **CS9207** requires interceptable methods to be ordinary member invocations.

Static abstract and virtual interface member errors and warnings

Article • 11/29/2023

The compiler generates the following errors for invalid declarations of static abstract or virtual members in interfaces:

- CS8920: *The interface cannot be used as type argument. Static member does not have a most specific implementation in the interface.*
- CS8921: *The parameter of a unary operator must be the containing type, or its type parameter constrained to it.*
- CS8922: *The parameter type for* `++` *or* `--` *operator must be the containing type, or its type parameter constrained to it.*
- CS8923: *The return type for* `++` *or* `--` *operator must either match the parameter type, or be derived from the parameter type, or be the containing type's type parameter constrained to it unless the parameter type is a different type parameter.*
- CS8924: *One of the parameters of a binary operator must be the containing type, or its type parameter constrained to it.*
- CS8925: *The first operand of an overloaded shift operator must have the same type as the containing type or its type parameter constrained to it*
- CS8926: *A static virtual or abstract interface member can be accessed only on a type parameter.*
- CS8928: *Type does not implement static interface member. The method cannot implement the interface member because it is not static.*
- CS8930: *Explicit implementation of a user-defined operator must be declared static*
- CS8931: *User-defined conversion in an interface must convert to or from a type parameter on the enclosing type constrained to the enclosing type*
- CS8932: *'UnmanagedCallersOnly' method cannot implement interface member in type*

These errors occur in three places in your code:

- When you declare an interface with static abstract or virtual members,
- When you declare a type that implements an interface with static abstract or virtual members, and
- When you access a static abstract or virtual method declared in an interface.

Errors in interface declaration

The following errors might occur when you declare an interface with `static abstract` or `static virtual` members:

- **CS8921**: *The parameter of a unary operator must be the containing type, or its type parameter constrained to it.*
- **CS8922**: *The parameter type for `++` or `--` operator must be the containing type, or its type parameter constrained to it.*
- **CS8923**: *The return type for `++` or `--` operator must either match the parameter type, or be derived from the parameter type, or be the containing type's type parameter constrained to it unless the parameter type is a different type parameter.*
- **CS8924**: *One of the parameters of a binary operator must be the containing type, or its type parameter constrained to it.*
- **CS8925**: *The first operand of an overloaded shift operator must have the same type as the containing type or its type parameter constrained to it*
- **CS8931**: *User-defined conversion in an interface must convert to or from a type parameter on the enclosing type constrained to the enclosing type*

All these rules are extensions of the rules for declaring overloaded operators. The distinction is that the parameter can be either the interface type, or the interface's type parameter if that type parameter is constrained to implement the interface for its type. For binary operators, only one parameter must satisfy this rule.

For example, `INumber<T>` can declare an `T operator++(T)` because `T` is constrained to implement `INumber<T>`.

To fix these errors, ensure that the parameters of any operators defined in the interface obey these rules. You can learn more in the language reference article on static abstract members in interfaces or in the tutorial to explore static abstract interface members.

Errors in type implementing interface declaration

The following errors might occur when you define a type that implements an interface with `static abstract` or `static virtual` methods:

- **CS8928**: *Type does not implement static interface member. The method cannot implement the interface member because it is not static.*
- **CS8930**: *Explicit implementation of a user-defined operator must be declared static*
- **CS8932**: *'UnmanagedCallersOnly' method cannot implement interface member in type*

These errors all indicate that you declared the method that implements a static abstract interface member incorrectly. These members must be declared `static`; they can't be instance members. Methods that implement interface members can't have the System.Runtime.InteropServices.UnmanagedCallersOnlyAttribute attribute applied to them.

Errors calling static abstract interface members

The following errors might occur when you attempt to call a member defined as a `static abstract` or `static virtual` member of an interface:

- **CS8920**: *The interface cannot be used as type argument. Static member does not have a most specific implementation in the interface.*
- **CS8926**: *A static virtual or abstract interface member can be accessed only on a type parameter.*

Calls to interface members declared as `static abstract` or `static virtual` must be resolved to at compile-time. They must resolve to a static member defined in a type that implements that interface. That means you must access those members using either a concrete type that implements the interface, or a type parameter that is constrained to implement the interface. To fix these errors, change the type used to access the static member.

Errors and warnings related to the `lock` statement and thread synchronization

Article • 05/07/2024

There are a few *errors* related to the `lock` statement and thread synchronization:

- CS0185: *'type' is not a reference type as required by the lock statement.*
- CS1996: *Cannot await in the body of a lock statement*
- CS9217: *A lock statement on a value of type 'System.Threading.Lock' cannot be used in async methods or async lambda expressions.*

In addition, the compiler might produce the following *warning* related to `lock` statements and thread synchronization:

- CS9216: *A value of type `System.Threading.Lock` converted to a different type will use likely unintended monitor-based locking in `lock` statement.*

lock statement errors

- **CS0185**: *'type' is not a reference type as required by the lock statement.*
- **CS1996**: *Cannot await in the body of a lock statement*
- **CS9217**: *A lock statement on a value of type 'System.Threading.Lock' cannot be used in async methods or async lambda expressions.*

These errors indicate that your code violates rules regarding the `lock` the statement:

- The object being `lock`ed must be a reference types. Value types aren't allowed.
- An `await` expression can't be used in the scope of a `lock` statement.
- The `lock` statement can't be used with `async` methods or lambda expressions. For this error, you can replace the type of object locked with a different type. The `lock` statement uses the Monitor API.

You must update your code to follow the rules of the `lock` statement.

lock warning

- **CS9216**: *A value of type `System.Threading.Lock` converted to a different type will use likely unintended monitor-based locking in `lock` statement.*

Beginning with C# 13, the lock generates specialized code when the target object is a `System.Threading.Lock` object. The compiler generates this warning when you're using a `Lock` object, but your code converts its type to something else. Therefore, the compiler generates the general locking code, not the locking code specific to the `Lock` type. For example:

```
C#

object lockObject = new System.Threading.Lock();

lock (lockObject) // CS9216
{
    // .. Your code
}
```

You should ensure the variable or expression represents the type of the `Lock` object.

Resolve warnings related to the dynamic type and dynamic binding

10/25/2025

This article covers the following compiler errors:

- CS1962: *The `typeof` operator cannot be used on the `dynamic` type.*
- CS1964: *Cannot apply dynamic conversion to an expression.*
- CS1965: *Cannot derive from the `dynamic` type.*
- CS1966: *Cannot derive from a constructed dynamic type.*
- CS1967: *Cannot use the `dynamic` type as a type constraint.*
- CS1968: *Cannot use a constructed dynamic type as a type constraint.*
- CS1969: *One or more types required to compile a dynamic expression cannot be found.*
- CS1970: *Do not use '`System.Runtime.CompilerServices.DynamicAttribute`'. Use the '`dynamic`' keyword instead.*
- CS1971: *The call to member needs to be dynamically dispatched, but cannot be because it is part of a base access expression. Consider casting the dynamic arguments or eliminating the base access.*
- CS1972: *The indexer access needs to be dynamically dispatched, but cannot be because it is part of a base access expression. Consider casting the dynamic arguments or eliminating the base access.*
- CS1973: *The dynamic argument type does not match the target parameter type for extension method.*
- CS1974: *Dynamic dispatch to a conditional method will fail at runtime.*
- CS1975: *The constructor call needs to be dynamically dispatched, but cannot be because it is part of a constructor initializer. Consider casting the dynamic arguments.*
- CS1976: *Cannot use a method group as an argument to a dynamically dispatched operation.*
- CS1977: *Cannot use a lambda expression as an argument to a dynamically dispatched operation.*
- CS1978: *Cannot use an expression as an argument to a dynamically dispatched operation.*
- CS1979: *Query expressions with a source or join sequence of type dynamic are not allowed.*
- CS1980: *Cannot define a class or member that uses 'dynamic' because the compiler required type is missing.*
- CS1981: *The '`is dynamic`' pattern is misleading. Use '`is object`' instead.*
- CS7083: *Expression must be implicitly convertible to '`System.Object`', or the type '`dynamic`' is not available.*
- CS8133: *Cannot deconstruct dynamic objects.*
- CS8364: *An argument to '`nameof`' cannot use any dynamic operation.*

- CS8416: *The async modifier cannot be used in the expression of a dynamic attribute.*
- CS9230: *Cannot perform a dynamic invocation on an expression with type.*

Using `dynamic` in type declarations and constraints

- **CS1962**: *The typeof operator cannot be used on the `dynamic` type.*
- **CS1965**: *Cannot derive from the `dynamic` type.*
- **CS1966**: *Cannot derive from a constructed dynamic type.*
- **CS1967**: *Cannot use the `dynamic` type as a type constraint.*
- **CS1968**: *Cannot use a constructed dynamic type as a type constraint.*
- **CS1970**: *Do not use '`System.Runtime.CompilerServices.DynamicAttribute`'. Use the '`dynamic`' keyword instead.*

The dynamic type provides late binding for operations at runtime. Use concrete types in contexts where the compiler needs type information at compile time, such as type declarations, constraints, inheritance, or reflection operations:

- Use concrete types for reflection operations. Use specific types instead of `dynamic` with the `typeof` operator (CS1962)
- Use concrete types for inheritance. Specify a concrete base class instead of `dynamic` (CS1965, CS1966):
- Use concrete type constraints. Specify concrete type constraints on generic parameters instead of `dynamic` (CS1967, CS1968):
- Use the `dynamic` keyword for variables. Always use the `dynamic` keyword to declare dynamic variables. Don't apply the `DynamicAttribute` directly (CS1970):

For more information about the `dynamic` type and its proper usage, see Using type dynamic.

Dynamic operation restrictions

- **CS1964**: *Cannot apply dynamic conversion to an expression.*
- **CS1971**: *The call to member needs to be dynamically dispatched, but cannot be because it is part of a base access expression. Consider casting the dynamic arguments or eliminating the base access.*
- **CS1972**: *The indexer access needs to be dynamically dispatched, but cannot be because it is part of a base access expression. Consider casting the dynamic arguments or eliminating the base access.*
- **CS1973**: *The dynamic argument type does not match the target parameter type for extension method.*

- **CS1975**: *The constructor call needs to be dynamically dispatched, but cannot be because it is part of a constructor initializer. Consider casting the dynamic arguments.*
- **CS1976**: *Cannot use a method group as an argument to a dynamically dispatched operation.*
- **CS1977**: *Cannot use a lambda expression as an argument to a dynamically dispatched operation.*
- **CS1978**: *Cannot use an expression as an argument to a dynamically dispatched operation.*
- **CS1979**: *Query expressions with a source or join sequence of type dynamic are not allowed.*
- **CS8133**: *Cannot deconstruct dynamic objects.*
- **CS8364**: *An argument to '`nameof`' cannot use any dynamic operation.*
- **CS8416**: *The async modifier cannot be used in the expression of a dynamic attribute.*
- **CS9230**: *Cannot perform a dynamic invocation on an expression with type.*

While dynamic binding provides flexibility at runtime, cast dynamic values to specific types when you need compile-time type information for certain operations.

- **Cast dynamic arguments before calling base members:** Cast dynamic arguments to their specific types before calling base members, indexers, or constructors (CS1971, CS1972, CS1975):

```C#
class Base
{
    public virtual void Method(object obj) { }
    public virtual int this[int index] => 0;
    public Base(int value) { }
}

class Derived : Base
{
    public Derived(dynamic value) : base((int)value) { }  // Cast before
calling base constructor

    public override void Method(object obj)
    {
        dynamic d = obj;
        base.Method((object)d);  // Cast before calling base method
    }

    public override int this[int index]
    {
        get
        {
            dynamic d = index;
            return base[(int)d];  // Cast before accessing base indexer
        }
```

```
    }
}
```

- **Cast to specific types before passing delegates or lambdas:** Cast the dynamic object to its concrete type before passing method groups, lambda expressions, or delegates (CS1976, CS1977, CS1978):

```C#
dynamic d = GetDynamicObject();

// Avoid:
d.ProcessData(Console.WriteLine);  // CS1976
d.ProcessData(x => x * 2);         // CS1977

// Recommended:
((IProcessor)d).ProcessData(Console.WriteLine);  // Cast first
((IProcessor)d).ProcessData(x => x * 2);         // Cast first
```

- **Use concrete types for LINQ queries:** Use a concrete type instead of `dynamic` for LINQ query sources and join sequences (CS1979):

```C#
dynamic data = GetData();

// Avoid:
var query = from item in data  // CS1979
            select item;

// Recommended:
IEnumerable<MyType> typedData = data;
var query = from item in typedData
            select item;
```

- **Access tuple elements individually:** Access tuple elements individually instead of using deconstruction with dynamic tuples (CS8133):

```C#
dynamic tuple = (1, 2);

// Avoid:
var (a, b) = tuple;  // CS8133

// Recommended:
var a = tuple.Item1;
var b = tuple.Item2;
```

- **Use compile-time expressions for operations requiring type information:** Use concrete types for operations that need compile-time type information (CS1964, CS1973, CS8364, CS8416, CS9230):

```
C#
```

```
dynamic value = GetValue();

// Avoid:
var name = nameof(value.Property);  // CS8364

// Recommended:
MyType typedValue = value;
var name = nameof(typedValue.Property);  // Use concrete type
```

For more information about dynamic binding and its limitations, see Using type dynamic.

Missing runtime support for dynamic

- **CS1969**: *One or more types required to compile a dynamic expression cannot be found. Are you missing a reference to 'Microsoft.CSharp.dll'?*
- **CS1980**: *Cannot define a class or member that uses 'dynamic' because the compiler required type cannot be found. Are you missing a reference?*
- **CS7083**: *Expression must be implicitly convertible to '`System.Object`', or the type '`dynamic`' is not available.*

The compiler needs types from the `System.Runtime` namespace and the Dynamic Language Runtime (DLR) to generate code for dynamic operations (CS1969, CS1980, CS7083). Ensure your project includes the necessary references. The required types are included in all modern .NET (.NET 5 and later) projects. For .NET Framework projects, add a reference to `Microsoft.CSharp.dll` in your project file.

For more information about dynamic type requirements, see Using type dynamic.

Dynamic dispatch warnings

- **CS1974**: *The dynamically dispatched call to method can fail at run-time because one or more applicable overloads are conditional methods.*
- **CS1981**: *The '`is dynamic`' pattern is misleading. The runtime type of the subexpression is never '`dynamic`'. Consider using '`object`' instead.*

When you call methods that have the `[Conditional]` attribute, avoid using dynamic dispatch. The compiler can't verify conditional method attributes with dynamic binding, which can cause

runtime failures (CS1974). Cast the dynamic expression to its actual type first:

```C#
dynamic d = GetObject();

// Avoid:
d.ConditionalMethod(); // CS1974 - can fail at runtime

// Recommended:
MyClass obj = (MyClass)d;
obj.ConditionalMethod(); // Compile-time checks ensure correctness
```

When you check whether a value is non-null, use `is object` instead of `is dynamic`. The `dynamic` keyword is a compile-time construct, and no object's runtime type is ever `dynamic` (CS1981):

```C#
// Avoid:
if (someValue is dynamic) // CS1981 - always evaluates to false
{
    // This code never executes
}

// Recommended:
if (someValue is object) // Correctly checks if non-null
{
    // This code executes for non-null values
}
```

For more information about dynamic dispatch and runtime behavior, see Using type dynamic.

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

Resolve errors and warnings in unsafe code constructs

10/24/2025

This article covers the following compiler errors:

- CS0193: *The * or -> operator must be applied to a data pointer*
- CS0196: *A pointer must be indexed by only one value*
- CS0208: *Cannot take the address of, get the size of, or declare a pointer to a managed type ('type')*
- CS0209: *The type of local declared in a fixed statement must be a pointer type*
- CS0210: *You must provide an initializer in a fixed or `using` statement declaration*
- CS0211: *Cannot take the address of the given expression*
- CS0212: *You can only take the address of an unfixed expression inside of a fixed statement initializer*
- CS0213: *You cannot use the fixed statement to take the address of an already fixed expression*
- CS0214: *Pointers and fixed size buffers may only be used in an unsafe context*
- CS0227: *Unsafe code may only appear if compiling with /unsafe*
- CS0233: *'identifier' does not have a predefined size, therefore sizeof can only be used in an unsafe context*
- CS0242: *The operation in question is undefined on void pointers*
- CS0244: *Neither '`is`' nor '`as`' is valid on pointer types*
- CS0254: *The right hand side of a fixed statement assignment may not be a cast expression*
- CS0459: *Cannot take the address of a read-only local variable*
- CS0821: *Implicitly typed locals cannot be fixed*
- CS1641: *A fixed size buffer field must have the array size specifier after the field name*
- CS1642: *Fixed size buffer fields may only be members of structs.*
- CS1656: *Cannot assign to 'variable' because it is a 'read-only variable type'*
- CS1663: *Fixed size buffer type must be one of the following: `bool`, `byte`, `short`, `int`, `long`, `char`, `sbyte`, `ushort`, `uint`, `ulong`, `float` or `double`*
- CS1665: *Fixed size buffers must have a length greater than zero*
- CS1666: *You cannot use fixed size buffers contained in unfixed expressions. Try using the fixed statement.*
- CS1708: *Fixed size buffers can only be accessed through locals or fields*
- CS1716: *Do not use '`System.Runtime.CompilerServices.FixedBuffer`' attribute. Use the 'fixed' field modifier instead.*
- CS1919: *Unsafe type 'type name' cannot be used in object creation.*
- CS4004: *Cannot `await` in an unsafe context*

- CS8812: *Cannot convert `&Method` group to non-function pointer type.*
- CS9123: *The '`&`' operator should not be used on parameters or local variables in async methods.*

Pointer operations and dereferencing

- **CS0193**: *The `*` or `->` operator must be applied to a data pointer*
- **CS0196**: *A pointer must be indexed by only one value*
- **CS0242**: *The operation in question is undefined on void pointers*

To use pointer operations correctly, follow the rules for dereferencing, indexing, and arithmetic operations. For more information, see Pointer types and Function pointers.

- Apply the `*` or `->` operator only to data pointers (**CS0193**). Don't use these operators with nonpointer types or function pointers. Function pointers can't be dereferenced in C#, unlike in C/C++.
- Index pointers with only one value (**CS0196**). Multidimensional indexing isn't supported on pointers.
- Avoid operations that are undefined on void pointers (**CS0242**). For example, don't increment a void pointer because the compiler doesn't know the size of the data being pointed to.

Pointer types and managed types

- **CS0208**: *Cannot take the address of, get the size of, or declare a pointer to a managed type ('type')*
- **CS0233**: *'identifier' does not have a predefined size, therefore sizeof can only be used in an unsafe context*

To work with pointers and the `sizeof` operator correctly, use unmanaged types and proper contexts. For more information, see Unmanaged types and the sizeof operator.

- Use pointers only with unmanaged types (**CS0208**). Don't take the address of, get the size of, or declare pointers to managed types. Managed types include reference types and structs that contain reference type fields or properties.
- Use the sizeof operator within an unsafe context when working with types whose size isn't a compile-time constant (**CS0233**).

Fixed buffers

- **CS0209**: *The type of local declared in a fixed statement must be a pointer type*

- **CS0210**: *You must provide an initializer in a fixed or `using` statement declaration*
- **CS0211**: *Cannot take the address of the given expression*
- **CS0212**: *You can only take the address of an unfixed expression inside of a fixed statement initializer*
- **CS0213**: *You cannot use the fixed statement to take the address of an already fixed expression*
- **CS0254**: *The right hand side of a fixed statement assignment may not be a cast expression*
- **CS0459**: *Cannot take the address of a read-only local variable*
- **CS0821**: *Implicitly typed locals cannot be fixed*
- **CS1656**: *Cannot assign to 'variable' because it is a 'read-only variable type'*

These errors occur when you use the fixed statement incorrectly. The `fixed` statement prevents the garbage collector from relocating a movable variable and declares a pointer to that variable. For more information, see Unsafe Code and Pointers.

To use the `fixed` statement correctly:

- Declare the variable as a pointer type (**CS0209**).
- Provide an initializer in the `fixed` statement declaration (**CS0210**).
- Take the address only of valid expressions: fields, local variables, and pointer indirection (**CS0211**). Don't take the address of computed expressions like the sum of two variables.
- Use the address-of operator on unfixed expressions only within the `fixed` statement initializer (**CS0212**).
- Don't use a `fixed` statement on already-fixed expressions (**CS0213**). Local variables and parameters in an `unsafe` method are already fixed on the stack.
- Don't use cast expressions on the right side of a `fixed` statement assignment (**CS0254**).
- Don't take the address of read-only local variables (**CS0459**). Variables in `foreach` loops, `using` statements, and `fixed` statements are read-only.
- Use explicit types instead of `var` in `fixed` statements (**CS0821**).
- Don't assign to variables in read-only contexts like `foreach` loops, `using` statements, or `fixed` statements (**CS1656**).

Unsafe context restrictions

- **CS0214**: *Pointers and fixed size buffers may only be used in an unsafe context*
- **CS0227**: *Unsafe code may only appear if compiling with /unsafe*
- **CS0244**: *Neither 'is' nor 'as' is valid on pointer types*
- **CS1919**: *Unsafe type 'type name' cannot be used in object creation*
- **CS4004**: *Cannot await in an unsafe context*

- **CS9123**: *The '&' operator should not be used on parameters or local variables in async methods*

These errors occur when you use unsafe code constructs without proper unsafe context or when you attempt operations that aren't allowed in unsafe code. For more information, see Unsafe Code and Pointers and the unsafe keyword.

To use unsafe code correctly:

- Mark methods, types, or code blocks that use pointers or fixed-size buffers with the `unsafe` keyword (**CS0214**).
- Enable the **AllowUnsafeBlocks** compiler option in your project settings when using the `unsafe` keyword (**CS0227**).
- Don't use the is or as operators with pointer types (**CS0244**). These type-testing operators aren't valid for pointers.
- Don't use the `new` operator to create pointer type instances (**CS1919**). To create objects in unmanaged memory, use interop to call native methods that return pointers.
- Keep unsafe code separate from async code (**CS4004**). Create separate methods for unsafe operations and call them from async methods.
- Don't use the address-of operator (`&`) on parameters or local variables in async methods (**CS9123**). The variable may not exist when the async operation completes.

Fixed-size buffers

- **CS1641**: *A fixed size buffer field must have the array size specifier after the field name*
- **CS1642**: *Fixed size buffer fields may only be members of structs*
- **CS1663**: *Fixed size buffer type must be one of the following: bool, byte, short, int, long, char, sbyte, ushort, uint, ulong, float or double*
- **CS1665**: *Fixed size buffers must have a length greater than zero*
- **CS1666**: *You cannot use fixed size buffers contained in unfixed expressions. Try using the fixed statement*
- **CS1708**: *Fixed size buffers can only be accessed through locals or fields*
- **CS1716**: *Do not use 'System.Runtime.CompilerServices.FixedBuffer' attribute. Use the 'fixed' field modifier instead*

These errors occur when you work with fixed-size buffers. Fixed-size buffers are arrays embedded directly in structs and are primarily used for interop scenarios. For more information, see Fixed-size buffers.

To declare and use fixed-size buffers correctly:

- Specify the array size after the field name using a positive integer constant (**CS1641**, **CS1665**).
- Declare fixed-size buffers only in structs, not in classes (**CS1642**). Use a regular array if you need the field in a class.
- Use one of the supported element types: `bool`, `byte`, `short`, `int`, `long`, `char`, `sbyte`, `ushort`, `uint`, `ulong`, `float`, or `double` (**CS1663**).
- Use a `fixed` statement to pin the containing struct before accessing the buffer (**CS1666**).
- Access fixed-size buffers only through locals or fields, not through intermediate expressions (**CS1708**).
- Use the `fixed` field modifier instead of the `System.Runtime.CompilerServices.FixedBuffer` attribute (**CS1716**).

Function pointers

- **CS8812**: *Cannot convert `&Method` group to non-function pointer type*

To obtain a function pointer, use the address-of operator with an explicit function pointer type cast. Don't use the address-of operator & to assign method groups to `void*` or other non-function pointer types. For more information, see Function pointers.

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

C# Warning waves

05/31/2025

New warnings and errors can be introduced in each release of the C# compiler. When new warnings could be reported on existing code, those warnings are introduced under an opt-in system referred to as a *warning wave*. The opt-in system means that you shouldn't see new warnings on existing code without taking action to enable them. When `<TreatWarningsAsErrors>true</TreatWarningsAsErrors>` is specified, enabled warning wave warnings generate errors. Warning wave 5 diagnostics were added in C# 9. Warning wave 6 diagnostics were added in C# 10. Warning wave 7 diagnostics were added in C# 11. Warning wave 8 diagnostics were added in C# 12. Warning wave 9 diagnostics were added in C# 13.

Beginning with the .NET 7 SDK (C# 11), the build system sets warning waves with the following rules:

- AnalysisLevel tracks the current TFM if not specified
- AnalysisLevel is set to latest if the current TFM is the 'latest' TFM (as defined by a property that we need to bump)
- WarningLevel should track the current TFM if not specified
- WarningLevel shouldn't override the user-provided value
- WarningLevel should be set to 4 if the project is a .NET Framework project

For SDKs earlier than .NET 7, AnalysisLevel always overwrote WarningLevel.

CS9123 - Taking address of local or parameter in async method can create a GC hole.

Warning wave 8

The `&` operator should not be used on parameters or local variables in async methods. The following code produces CS9123:

C#

```
public static async Task LogValue()
{
    int x = 1;
    unsafe {
        int* y = &x;
        Console.WriteLine(*y);
    }
```

```
    await Task.Delay(1000);
}
```

CS8981 - The type name only contains lower-cased ascii characters.

Warning wave 7

Any new keywords added for C# will be all lower-case ASCII characters. This warning ensures that none of your types conflict with future keywords. The following code produces CS8981:

C#

```
public class lowercasename
{
}
```

You can address this warning by renaming the type to include at least one non-lower case ASCII character, such as an upper case character, a digit, or an underscore.

CS8826 - Partial method declarations have signature differences.

Warning wave 6

This warning corrects some inconsistencies in reporting differences between partial method signatures. The compiler always reported an error when the partial method signatures created different CLR signatures. Now, the compiler reports CS8826 when the signatures are syntactically different C#. Consider the following partial class:

C#

```
public partial class PartialType
{
    public partial void M1(int x);

    public partial T M2<T>(string s) where T : struct;

    public partial void M3(string s);


    public partial void M4(object o);
    public partial void M5(dynamic o);
```

```
    public partial void M6(string? s);
}
```

The following partial class implementation generates several examples of CS8626:

```
C#

public partial class PartialType
{
    // Different parameter names:
    public partial void M1(int y) { }

    // Different type parameter names:
    public partial TResult M2<TResult>(string s) where TResult : struct =>
default;

    // Relaxed nullability
    public partial void M3(string? s) { }


    // Mixing object and dynamic
    public partial void M4(dynamic o) { }

    // Mixing object and dynamic
    public partial void M5(object o) { }

    // Note: This generates CS8611 (nullability mismatch) not CS8826
    public partial void M6(string s) { }
}
```

> **Note**
>
> If the implementation of a method uses a non-nullable reference type when the other declaration accepts nullable reference types, CS8611 is generated instead of CS8826.

To fix any instance of these warnings, ensure the two signatures match.

CS7023 - A static type is used in an 'is' or 'as' expression.

Warning wave 5

The `is` and `as` expressions always return `false` for a static type because you can't create instances of a static type. The following code produces CS7023:

```
C#
```

```
static class StaticClass
{
    public static void Thing() { }
}

void M(object o)
{
    // warning: cannot use a static type in 'is' or 'as'
    if (o is StaticClass)
    {
        Console.WriteLine("Can't happen");
    }
    else
    {
        Console.WriteLine("o is not an instance of a static class");
    }
}
```

The compiler reports this warning because the type test can never succeed. To correct this warning, remove the test and remove any code executed only if the test succeeded. In the preceding example, the `else` clause is always executed. You can replace that method body with that single line:

C#

```
Console.WriteLine("o is not an instance of a static class");
```

CS8073 - The result of the expression is always 'false' (or 'true').

Warning wave 5

The `==` and `!=` operators always return `false` (or `true`) when comparing an instance of a `struct` type to `null`. The following code demonstrates this warning. Assume `S` is a `struct` that defines `operator ==` and `operator !=`:

C#

```
class Program
{
    public static void M(S s)
    {
        if (s == null) { } // CS8073: The result of the expression is always
'false'
        if (s != null) { } // CS8073: The result of the expression is always
'true'
```

```
    }
}

struct S
{
    public static bool operator ==(S s1, S s2) => s1.Equals(s2);
    public static bool operator !=(S s1, S s2) => !s1.Equals(s2);
    public override bool Equals(object? other)
    {
        // Implementation elided
        return false;
    }
    public override int GetHashCode() => 0;

    // Other details elided...
}
```

To fix this error, remove the null check and code that would execute if the object is `null`.

CS8848 - Operator 'from' can't be used here due to precedence. Use parentheses to disambiguate.

Warning wave 5

The following examples demonstrate this warning. The expression binds incorrectly because of the precedence of the operators.

C#

```
bool b = true;
var source = new Src();
b = true;
source = new Src();
var a = b && from c in source select c;
Console.WriteLine(a);

var indexes = new Src2();
int[] array = { 1, 2, 3, 4, 5, 6, 7 };
var range = array[0..from c in indexes select c];
```

To fix this error, put parentheses around the query expression:

C#

```
bool b = true;
var source = new Src();
b = true;
source = new Src();
```

```
var a = b && (from c in source select c);
Console.WriteLine(a);

var indexes = new Src2();
int[] array = { 1, 2, 3, 4, 5, 6, 7 };
var range = array[0..(from c in indexes select c)];
```

Members must be fully assigned. Use of unassigned variable (CS8880, CS8881, CS8882, CS8883, CS8884, CS8885, CS8886, CS8887)

Warning wave 5

Several warnings improve the definite assignment analysis for `struct` types declared in imported assemblies. All these new warnings are generated when a struct in an imported assembly includes an inaccessible field (usually a `private` field) of a reference type, as shown in the following example:

C#

```
public struct Struct
{
    private string data = String.Empty;
    public Struct() { }
}
```

The following examples show the warnings generated from the improved definite assignment analysis:

- CS8880: Auto-implemented property 'Property' must be fully assigned before control is returned to the caller.
- CS8881: Field 'field' must be fully assigned before control is returned to the caller.
- CS8882: The `out` parameter 'parameter' must be assigned to before control leaves the current method.
- CS8883: Use of possibly unassigned auto-implemented property 'Property'.
- CS8884: Use of possibly unassigned field 'Field'
- CS8885: The 'this' object can't be used before all its fields have been assigned.
- CS8886: Use of unassigned output parameter 'parameterName'.
- CS8887: Use of unassigned local variable 'variableName'

C#

```
public struct DefiniteAssignmentWarnings
{
    // CS8880
    public Struct Property { get; }
    // CS8881
    private Struct field;

    // CS8882
    public void Method(out Struct s)
    {

    }

    public DefiniteAssignmentWarnings(int dummy)
    {
        // CS8883
        Struct v2 = Property;
        // CS8884
        Struct v3 = field;
        // CS8885:
        DefiniteAssignmentWarnings p2 = this;
    }

    public static void Method2(out Struct s1)
    {
        // CS8886
        var s2 = s1;
        s1 = default;
    }

    public static void UseLocalStruct()
    {
        Struct r1;
        var r2 = r1;
    }
}
```

You can fix any of these warnings by initializing or assigning the imported struct to its default value:

C#

```
public struct DefiniteAssignmentNoWarnings
{
    // CS8880
    public Struct Property { get; } = default;
    // CS8881
    private Struct field = default;

    // CS8882
    public void Method(out Struct s)
    {
```

```
        s = default;
    }

    public DefiniteAssignmentNoWarnings(int dummy)
    {
        // CS8883
        Struct v2 = Property;
        // CS8884
        Struct v3 = field;
        // CS8885:
        DefiniteAssignmentNoWarnings p2 = this;
    }

    public static void Method2(out Struct s1)
    {
        // CS8886
        s1 = default;
        var s2 = s1;
    }

    public static void UseLocalStruct()
    {
        Struct r1 = default;
        var r2 = r1;
    }
}
```

CS8892 - Method will not be used as an entry point because a synchronous entry point 'method' was found.

Warning wave 5

This warning is generated on all async entry point candidates when you have multiple valid entry points, including one or more synchronous entry point.

The following example generates CS8892:

C#

```
public static void Main()
{
    RunProgram();
}

// CS8892
public static async Task Main(string[] args)
{
```

```
    await RunProgramAsync();
}
```

> **Note**
>
> The compiler always uses the synchronous entry point. In case there are multiple synchronous entry points, you get a compiler error.

To fix this warning, remove or rename the asynchronous entry point.

CS8897 - Static types can't be used as parameters

Warning wave 5

Members of an interface can't declare parameters whose type is a static class. The following code demonstrates both CS8897 and CS8898:

C#

```
public static class Utilities
{
    // elided
}

public interface IUtility
{
    // CS8897
    public void SetUtility(Utilities u);

    // CS8898
    public Utilities GetUtility();
}
```

To fix this warning, change the parameter type or remove the method.

CS8898 - static types can't be used as return types

Warning wave 5

Members of an interface can't declare a return type that is a static class. The following code demonstrates both CS8897 and CS8898:

C#

```
public static class Utilities
{
    // elided
}

public interface IUtility
{
    // CS8897
    public void SetUtility(Utilities u);

    // CS8898
    public Utilities GetUtility();
}
```

To fix this warning, change the return type or remove the method.

Compiler Error CS0001

Article • 09/15/2021

Internal compiler error

Try to determine whether the compiler is failing because of its inability to parse unexpected syntax. If you receive this error repeatedly, please contact Microsoft.

Compiler Error CS0003

Article • 09/15/2021

Out of memory

The compiler was unable to allocate enough virtual memory to complete compilation. Close all unnecessary applications and compile again.

You may also want to increase the pagefile size and make sure that you have free disk space.

This error may also appear as the result of having mismatched versions of the Windows SDK and the C# compiler or one or more corrupted files that support the C# compiler; reinstall Visual Studio.

Compiler Error CS0004

Article • 09/15/2021

Warning treated as error

You compiled with the `/warnaserror` compiler option, which causes the compiler to generate warnings as errors.

Compiler Error CS0005

Article • 09/15/2021

Compiler option 'compiler_option' must be followed by an argument

Some compiler options require parameters. If you do not pass the arguments required by the compiler option, CS0005 is generated.

For more information, see C# Compiler Options.

Compiler Error CS0006

Article • 09/15/2021

Metadata file 'dll_name' could not be found

The program was compiled and explicitly passed the name of a file that contained metadata; however, the .dll was not found. For more information, see **References** (C# Compiler Options).

Compiler Error CS0007

Article • 09/15/2021

Unexpected common language runtime initialization error — 'description'

This error occurs if the runtime could not be loaded. This could occur if the version of the common language runtime that the compiler attempts to load is not present on the machine, or if the common language runtime installation or configuration is corrupt.

This can happen if the *csc.exe.config* file was changed. This file is configured during setup and should not be changed. If there is a possibility that the *csc.exe.config* file was changed, check the file to make sure that the version of the runtime specified in the file is present on the machine. If the correct version is present, it may be corrupted. Reinstall the common language runtime.

Compiler Error CS0008

Article • 09/15/2021

Unexpected error reading metadata from file 'file' — 'description'

A DLL was successfully opened for retrieving metadata but is corrupted, such that data could not be read from it. For more information, see **References** (C# Compiler Options).

Compiler Error CS0009

Article • 09/15/2021

Metadata file 'file' could not be opened — 'description'

The file specified with the **References** compiler option does not contain valid metadata.

Compiler Error CS0010

Article • 09/15/2021

Unexpected fatal error -- 'error'.

This error is generated when something completely unexpected occurs to stop compilation.

To correct this error

Recompile another project. If you receive the same error, try reinstalling Visual Studio. If you receive this error, send an error report to Microsoft.

Compiler Error CS0011

Article • 09/15/2021

The base class or interface 'class' in assembly 'assembly' referenced by type 'type' could not be resolved

A class that was imported from a file with **/reference**, is derived from a class or implements an interface that is not found. This can occur if a required DLL is not also included in the compilation with **/reference**.

For more information, see Add Reference Dialog Box and **References** (C# Compiler Options).

Examples

C#

```
// CS0011_1.cs
// compile with: /target:library

public class Outer
{
   public class B { }
}
```

The second file creates a DLL that defines a class `C` that is derived from the class `B` that was created in the previous example.

C#

```
// CS0011_2.cs
// compile with: /target:library /reference:CS0011_1.dll
// post-build command: del /f CS0011_1.dll
public class C : Outer.B {}
```

The third file replaces the DLL created by the first step, and omits the definition of the inner class `B`.

C#

```
// CS0011_3.cs
// compile with: /target:library /out:cs0011_1.dll
public class Outer {}
```

Finally, the fourth file references the class `C` defined in the second example, which is derived from class `B`, and which is now missing.

The following sample generates CS0011.

C#

```
// CS0011_4.cs
// compile with: /reference:CS0011_1.dll /reference:CS0011_2.dll
// CS0011 expected

class M
{
   public static void Main()
   {
      C c = new C();
   }
}
```

Compiler Error CS0013

Article • 09/15/2021

Unexpected error writing metadata to file 'ModelStore.dll' -- 'No logical space left to create more user strings.

The .NET runtime did not emit metadata. Check to make sure that the path is correct and that the disk is not full. If the problem persists, you may have to repair or reinstall Visual Studio and/or .NET.

Compiler Error CS0014

Article • 09/15/2021

Required file 'file' could not be found

A file is needed by the compiler but is not on the system. Make sure that the path is correct. If the file is a Visual Studio system file, then you may need to repair your installation or else remove and reinstall Visual Studio completely.

Compiler Error CS0015

Article • 09/15/2021

The name of type 'type' is too long

The fully qualified name of a user-defined type must have fewer than 1024 characters, including the periods.

The following sample generates CS0015:

C#

```
// CS0015.cs

// Remove a C from one of the namespace names or the class name to resolve
the error by
// reducing the number of characters in the qualified class name to fewer
than 1024.

namespace CC
{
    // The following namespace name has 510 characters.
    namespace
CCC510charsCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
    {
        // The following class name has 510 characters. The qualified class
name,
        // CC.CCC510chars…C.CCCC510chars…C (namespace.namespace.class), has
1024
        // characters, which causes compiler error CS0015.
        public class
CCCC510charsCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
CCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCCC
        {
            public static void Main()
            {
            }
        }
```

```
    }
}
```

Compiler Error CS0016

Article • 09/15/2021

Could not write to output file 'file' — 'reason'

The compiler could not write to an output file. Check the path to the file to make sure it exists. If a previously built file is already at the location, make sure it is writeable, and that no process currently has a lock on the file. For example, make sure your executable is not running when you attempt to build.

Compiler Error CS0017

07/22/2025

Program 'output file name' has more than one entry point defined. Compile with /main to specify the type that contains the entry point.

A program can only have one Main method.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

To resolve this error, you can either delete all Main methods in your code, except one, or you can use the **StartupObject** compiler option to specify which Main method you want to use.

The following sample generates CS0017:

C#

```
// CS0017.cs
// compile with: /target:exe
public class clx
{
   static public void Main()
   {
   }
}

public class cly
{
   public static void Main()   // CS0017, delete one Main or use /main
   {
   }
}
```

Compiler Error CS0019

Article • 09/15/2021

Operator 'operator' cannot be applied to operands of type 'type' and 'type'

A binary operator is applied to data types that do not support it. For example, you cannot use the || operator on strings, you cannot use +, -, <, or > operators on bool variables, and you cannot use the == operator with a `struct` type unless the type explicitly overloads that operator.

You can overload an operator to make it support operands of certain types. For more information, see Operator overloading.

Example 1

In the following example, CS0019 is generated in three places because bool in C# is not convertible to int. CS0019 is also generated when the subtraction operator `-` is applied to a string. The addition operator `+` can be used with string operands because that operator is overloaded by the `String` class to perform string concatenation.

C#

```
static void Main()
{
    bool result = true;
    if (result > 0) //CS0019
    {
        // Do something.
    }

    int i = 1;
    // You cannot compare an integer and a boolean value.
    if (i == true) //CS0019
    {
        //Do something...
    }

    string s = "Just try to subtract me.";
    float f = 100 - s; // CS0019
}
```

Example 2

In the following example, conditional logic must be specified outside the ConditionalAttribute. You can pass only one predefined symbol to the ConditionalAttribute.

The following sample generates CS0019:

C#

```
// CS0019_a.cs
// compile with: /target:library
using System.Diagnostics;

public class MyClass
{
   [ConditionalAttribute("DEBUG" || "TRACE")]   // CS0019
   public void TestMethod() {}

   // OK
   [ConditionalAttribute("DEBUG"), ConditionalAttribute("TRACE")]
   public void TestMethod2() {}
}
```

See also

- C# operators

Compiler Error CS0020

Article • 09/15/2021

Division by constant zero

An expression uses a literal (not variable) value of zero in the denominator of a division operation. Division by zero is not defined, and is therefore invalid.

The following sample generates CS0020:

C#

```
// CS0020.cs
namespace x
{
   public class b
   {
      public static void Main()
      {
         1 / 0;   // CS0020
      }
   }
}
```

See also

- Arithmetic operators

Compiler Error CS0021

Article • 09/15/2021

Cannot apply indexing with [] to an expression of type 'type'

An attempt was made to access a value through an indexer on a data type that does not support Indexers.

You may get CS0021 when trying to use an indexer in a C++ assembly. In this case, decorate the C++ class with the `DefaultMember` attribute so the C# compiler knows which indexer is the default. The following sample generates CS0021.

Example

The following C++ example compiles to a .dll file. Note that the DefaultMember attribute is commented out in order to generate the error.

C++

```
// CPP0021.cpp
// compile with: /clr /LD
using namespace System::Reflection;
// Uncomment the following line to resolve
//[DefaultMember("myItem")]
public ref class MyClassMC
{
        public:
        property int myItem[int]
        {
            int get(int i){  return 5; }
            void set(int i, int value) {}
        }
};
```

The following C# example calls the .dll file. It attempts to access the class via an indexer, but because no member was declared as the default indexer, the error is generated. You can address the error by uncommenting the line `[DefaultMember("myItem")]` in the .cpp file in the preceding example.

C#

```
// CS0021.cs
// compile with: /reference:CPP0021.dll
public class MyClass
{
```

```
    public static void Main()
    {
        MyClassMC myMC = new MyClassMC();
        int j = myMC[1]; // CS0021
    }
}
```

Compiler Error CS0023

Article • 10/27/2021

Operator 'operator' cannot be applied to operand of type 'type'

An attempt was made to apply an operator to a variable whose type was not designed to work with the operator. For more information, see Types and C# Operators.

The following sample generates CS0023:

C#

```
// CS0023.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         string s = "hello";
         s = -s;   // CS0023, minus operator not allowed on strings
      }
   }
}
```

Compiler Error CS0025

Article • 09/15/2021

Standard library file 'file' could not be found

A file that is needed by the compiler was not found. Check that the path is correct and that the file exists.

If the file is a Visual Studio system file, you may need to repair your Visual Studio installation, or reinstall it completely.

Compiler Error CS0026

Article • 09/15/2021

Keyword 'this' is not valid in a static property, static method, or static field initializer

The this keyword refers to an object, which is an instance of a type. Since static methods are independent of any instance of the containing class, the "this" keyword is meaningless and is therefore not allowed. For more information, see Static Classes and Static Class Members and Objects.

Example

The following example generates CS0026:

C#

```
// CS0026.cs
public class A
{
   public static int i = 0;

   public static void Main()
   {
// CS0026
      this.i = this.i + 1;
      // Try the following line instead:
      // i = i + 1;
   }
}
```

Compiler Error CS0027

Article • 09/15/2021

Keyword 'this' is not available in the current context

The this keyword was found outside of a property, method, or constructor.

To fix this error, either modify the statement to eliminate use of the `this` keyword, and/or move part or all of the statement inside a property, method, or constructor.

The following example generates CS0027:

C#

```
using System;
using System.Collections.Generic;
using System.Text;

namespace ConsoleApplication3
{
    class MyClass
    {

        int err1 = this.Fun() + 1;  // CS0027

        public int Fun()
        {
            return 10;
        }

        public void Test()
        {
            // valid use of this
            int err = this.Fun() + 1;
            Console.WriteLine(err);
        }

        public static void Main()
        {
            MyClass c = new MyClass();
            c.Test();
        }
    }
}
```

Compiler Error CS0029

Article • 09/15/2021

Cannot implicitly convert type 'type' to 'type'

The compiler requires an explicit conversion. For example, you may need to cast an r-value to be the same type as an l-value. Or, you must provide conversion routines to support certain operator overloads.

Conversions must occur when assigning a variable of one type to a variable of a different type. When making an assignment between variables of different types, the compiler must convert the type on the right-hand side of the assignment operator to the type on the left-hand side of the assignment operator. Take the following the code:

C#

```
int i = 50;
long lng = 100;
i = lng;
```

`i = lng;` makes an assignment, but the data types of the variables on the left and right-hand side of the assignment operator don't match. Before making the assignment the compiler is implicitly converting the variable `lng`, which is of type long, to an int. This is implicit because no code explicitly instructed the compiler to perform this conversion. The problem with this code is that this is considered a narrowing conversion, and the compiler does not allow implicit narrowing conversions because there could be a potential loss of data.

A narrowing conversion exists when converting to a data type that occupies less storage space in memory than the data type we are converting from. For example, converting a long to an int would be considered a narrowing conversion. A long occupies 8 bytes of memory while an int occupies 4 bytes. To see how data loss can occur, consider the following sample:

C#

```
int i = 50;
long lng = 3147483647;
i = lng;
```

The variable `lng` now contains a value that cannot be stored in the variable `i` because it is too large. If we were to convert this value to an int type we would be losing some of

our data and the converted value would not be the same as the value before the conversion.

A widening conversion would be the opposite of a narrowing conversion. With widening conversions, we are converting to a data type that occupies more storage space in memory than the data type we are converting from. Here is an example of a widening conversion:

```
C#

int i = 50;
long lng = 100;
lng = i;
```

Notice the difference between this code sample and the first. This time the variable `lng` is on the left-hand side of the assignment operator, so it is the target of our assignment. Before the assignment can be made, the compiler must implicitly convert the variable `i`, which is of type int, to type long. This is a widening conversion since we are converting from a type that occupies 4 bytes of memory (an int) to a type that occupies 8 bytes of memory (a long). Implicit widening conversions are allowed because there is no potential loss of data. Any value that can be stored in an int can also be stored in a long.

We know that implicit narrowing conversions are not allowed, so to be able to compile this code we need to explicitly convert the data type. Explicit conversions are done using casting. Casting is the term used in C# to describe converting one data type to another. To get the code to compile we would need to use the following syntax:

```
C#

int i = 50;
long lng = 100;
i = (int) lng;   // Cast to int.
```

The third line of code tells the compiler to explicitly convert the variable `lng`, which is of type long, to an int before making the assignment. Remember that with a narrowing conversion, there is a potential loss of data. Narrowing conversions should be used with caution and even though the code will compile you may get unexpected results at run-time.

This discussion has only been for value types. When working with value types you work directly with the data stored in the variable. However, .NET also has reference types. When working with reference types you are working with a reference to a variable, not the actual data. Examples of reference types would be classes, interfaces and arrays. You

cannot implicitly or explicitly convert one reference type to another unless the compiler allows the specific conversion or the appropriate conversion operators are implemented.

The following sample generates CS0029:

C#

```
// CS0029.cs
public class MyInt
{
    private int x = 0;

    // Uncomment this conversion routine to resolve CS0029.
    /*
    public static implicit operator int(MyInt i)
    {
        return i.x;
    }
    */

    public static void Main()
    {
        var myInt = new MyInt();
        int i = myInt; // CS0029
    }
}
```

See also

- User-defined conversion operators

Compiler Error CS0030

Article • 09/15/2021

Cannot convert type 'type' to 'type'

There is no predefined conversion between types. You can define a custom conversion between those types. For more information, see User-defined conversion operators.

The following sample generates CS0030:

C#

```
// CS0030.cs
namespace x
{
   public class iii
   {
      /*
      public static implicit operator iii(int aa)
      {
         return null;
      }

      public static implicit operator int(iii aa)
      {
         return 0;
      }
      */

      public static iii operator ++(iii aa)
      {
         return (iii)0;   // CS0030
         // uncomment the conversion routines to resolve CS0030
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0031

Article • 08/24/2022

Constant value 'value' cannot be converted to a 'type'.

An attempt was made to assign a value to a variable whose type cannot store the value. For more information, see Types.

The following sample generates CS0031 in both checked and unchecked contexts:

C#

```
// CS0031.cs
namespace CS0031
{
    public class Program
    {
        public static void Main()
        {
            int num = (int)2147483648M; //CS0031
            // Try using a larger numeric type instead.
            // long num = (long)2147483648M; //CS0031

            const decimal d = -10M; // Decimal literal
            unchecked
            {
                const byte b = (byte)d; // CS0031
                // For small values try using a signed byte instead.
                // const sbyte b = (sbyte)d;
            }
        }
    }
}
```

See also

- unchecked

Compiler Error CS0036

Article • 09/15/2021

An out parameter cannot have the '[In]' attribute

Currently, the **In** attribute is not permitted on an out parameter.

The following sample generates CS0036:

C#

```
// CS0036.cs

using System;
using System.Runtime.InteropServices;

public class MyClass
{
   public static void TestOut([In] out char TestChar)   // CS0036
   // try the following line instead
   // public static void TestOut(out char TestChar)
   {
      TestChar = 'b';
      Console.WriteLine(TestChar);
   }

   public static void Main()
   {
      char i;           //variable to receive the value
      TestOut(out i);   // the arg must be passed as out
      Console.WriteLine(i);
   }
}
```

Compiler Error CS0037

Article • 09/15/2021

Cannot convert null to 'type' because it is a non-nullable value type

The compiler cannot assign `null` to a value type; `null` can only be assigned to a reference type or to a nullable value type. struct is a value type.

The following sample generates CS0037:

C#

```
// CS0037.cs
public struct s
{
}

class a
{
   public static void Main()
   {
      int i = null;   // CS0037
      s ss = null;    // CS0037
   }
}
```

Compiler Error CS0038

Article • 09/15/2021

Cannot access a nonstatic member of outer type 'type1' via nested type 'type2'

A field in a class is not automatically available to a nested class. To be available to a nested class, the field must be static. Otherwise, you must create an instance of the outer class. For more information, see Nested Types.

The following sample generates CS0038:

C#

```
// CS0038.cs
class OuterClass
{
    public int count;
    // Try the following line instead.
    // public static int count;

    class InnerClass
    {
        void Func()
        {
            // or, create an instance
            // OuterClass class_inst = new OuterClass();
            // int count2 = class_inst.count;
            int count2 = count;   // CS0038
        }
    }

    public static void Main()
    {
    }
}
```

Compiler Error CS0039

Article • 09/15/2021

Cannot convert type 'type1' to 'type2' via a reference conversion, boxing conversion, unboxing conversion, wrapping conversion, or null type conversion

A conversion with the as operator is allowed by inheritance, reference conversions, and boxing conversions.

Example

The following example generates CS0039:

C#

```
using System;

class A { }
class B : A { }
class C : A { }

class Example
{
    static void Main()
    {
        C c;

        // This compiles, because
        // there is an explicit reference conversion from type A to type C.
        A a = new C();
        c = a as C;

        // This generates CS0039, because
        // there is no implicit or explicit reference conversion between B
and C types.
        B b = new B();
        c = b as C;  // CS0039
    }
}
```

Compiler Error CS0040

Article • 09/15/2021

Unexpected error creating debug information file — 'reason'

This error can occur when using the **DebugType** compiler option and indicates that the compiler was unable to write to the .pdb file. Possible resolutions to this error include reinstalling Visual Studio, ensuring that the compiler has write access to a file or directory, or not compiling with /debug.

Compiler Error CS0041

Article • 09/15/2021

Unexpected error writing debug information -- '{error}'

Compiler Error CS0042

Article • 09/15/2021

Unexpected error creating debug information file 'file' — 'reason'

The compiler could not create the debug information; *reason* contains additional information on the error condition.

Compiler Error CS0043

Article • 09/15/2021

PDB file 'file' has an incorrect or out-of-date format. Delete it and rebuild.

Delete the .pdb file for this compilation and recompile.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0050

08/15/2025

Inconsistent accessibility: return type 'type' is less accessible than method 'method'

The return type and each of the types referenced in the formal parameter list of a method must be at least as accessible as the method itself. This includes type arguments of generic types used in the return type or parameters. For more information, see Access Modifiers.

Examples

The following sample generates CS0050 because no accessibility modifier is supplied for `MyClass`, and its accessibility therefore defaults to `private`:

C#

```
// CS0050.cs
class MyClass // Accessibility defaults to private.
// Try the following line instead.
// public class MyClass
{
}

public class MyClass2
{
    public static MyClass MyMethod()   // CS0050
    {
        return new MyClass();
    }

    public static void Main() { }
}
```

CS0050 can also occur when a generic type's type argument is less accessible than the method:

C#

```
// CS0050_Generic.cs
using System.Collections.ObjectModel;

internal class CeisData // Internal class
{
    public string Name { get; set; }
}

public class MyClass
{
```

```
    public static ObservableCollection<CeisData> BuildCeis()   // CS0050
    {
        return new ObservableCollection<CeisData>();
    }
}
```

To fix this error, make the type argument at least as accessible as the method:

C#

```
// Fixed version
using System.Collections.ObjectModel;

public class CeisData // Now public
{
    public string Name { get; set; }
}

public class MyClass
{
    public static ObservableCollection<CeisData> BuildCeis()   // OK
    {
        return new ObservableCollection<CeisData>();
    }
}
```

Compiler Error CS0051

08/07/2025

Inconsistent accessibility: parameter type 'type' is less accessible than method 'method'

This error occurs when you declare a method (including constructors) with a parameter type that's less accessible than the method itself. For example, you might have a public constructor that uses an internal or private class as a parameter type.

The most common scenario is when you define a public method but one of its parameter types is internal or private. This creates an inconsistency because external code can see the method but can't access the types needed to call it.

How to troubleshoot this error

1. **Identify the parameter type causing the issue**: Look at the error message to see which parameter type is less accessible.
2. **Check the accessibility of the parameter type**: Right-click on the parameter type in your IDE and select "Go to Definition" (or press F12) to see how it's declared.
3. **Compare accessibility levels**: Ensure the parameter type is at least as accessible as the method that uses it.

Examples

Example 1: Public method with private parameter type

The following sample generates CS0051 because the method `F` is public but the parameter type `B` is private:

C#

```
// CS0051.cs
public class A
{
    // B is implicitly private here.
    class B
    {
    }

    public static void F(B b)  // CS0051
    {
    }
```

```
    public static void Main()
    {
    }
}
```

Example 2: Public constructor with internal parameter type

This is a common scenario where you have a public constructor but the parameter type is internal:

C#

```
// Another file or assembly.
internal class DatabaseConfiguration
{
    public string ConnectionString { get; set; }
}

// In your main class.
public class DataService
{
    // This causes CS0051 because the constructor is public.
    // but DatabaseConfiguration is internal.
    public DataService(DatabaseConfiguration config)  // CS0051
    {
        // Implementation.
    }
}
```

To correct this error

Choose one of the following approaches:

1. **Make the parameter type more accessible**: Change the parameter type to match or exceed the accessibility of the method:

   C#

   ```
   public class A
   {
       // Make B public to match the accessibility of method F.
       public class B
       {
       }

       public static void F(B b)  // Now works correctly
       {
   ```

```
    }
}
```

2. **Reduce the accessibility of the method**: Make the method less accessible to match the parameter type:

```
C#

public class A
{
    class B  // B remains private.
    {
    }

    // Make F internal or private to match B's accessibility.
    internal static void F(B b)  // Now works correctly
    {
    }
}
```

3. **Use a more accessible interface or base class**: Instead of using the less accessible type directly, use a public interface or base class:

```
C#

public interface IConfiguration
{
    string ConnectionString { get; }
}

internal class DatabaseConfiguration : IConfiguration
{
    public string ConnectionString { get; set; }
}

public class DataService
{
    // Use the public interface instead.
    public DataService(IConfiguration config)  // Works correctly
    {
        // Implementation.
    }
}
```

See also

- Access Modifiers
- Accessibility Levels

- Constructors

Compiler Error CS0052

Article • 09/15/2021

Inconsistent accessibility: field type 'type' is less accessible than field 'field'

The type of a field cannot be less accessible than the field itself because all public constructs must return a publicly accessible object.

Example

The following sample generates CS0052:

C#

```
// CS0052.cs
public class MyClass2
{
    // The following line causes an error because the field, M, is declared
    // as public, but the type, MyClass, is private. Therefore the type is
    // less accessible than the field.
    public MyClass M;   // CS0052

    private class MyClass
    {
    }
    // One way to resolve the error is to change the accessibility of the
type
    // to public.
    //public class MyClass
    // Another solution is to change the accessibility of the field to
private.
    // private MyClass M;
}

public class MainClass
{
    public static void Main()
    {
    }
}
```

See also

- C# Keywords
- Access Modifiers
- Accessibility Levels

- Modifiers

Compiler Error CS0053

Article • 09/15/2021

Inconsistent accessibility: property type 'type' is less accessible than property 'property'

A public construct must return a publicly accessible object. For more information, see Access Modifiers.

The following sample generates CS0053:

C#

```
// CS0053.cs
class MyClass //defaults to private accessibility
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public MyClass myProperty   // CS0053
   {
      get
      {
         return new MyClass();
      }
      set
      {
      }
   }
}

public class MyClass3
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0054

Article • 09/15/2021

Inconsistent accessibility: indexer return type 'type' is less accessible than indexer 'indexer'

A public construct must return a publicly accessible object. For more information, see Access Modifiers.

The following sample generates CS0054:

C#

```
// CS0054.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public class MyClass3
{
   public MyClass this[int i]   // CS0054
   {
      get
      {
         return new MyClass();
      }
   }
}

public class MyClass2
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0055

Article • 09/15/2021

Inconsistent accessibility: parameter type 'type' is less accessible than indexer 'indexer'

A public construct must return a publicly accessible object. For more information, see Access Modifiers.

The following sample generates CS0055:

C#

```
// CS0055.cs
class MyClass //defaults to private accessibility
// try the following line instead
// public class MyClass
{
}

public class MyClass2
{
   public int this[MyClass myClass]   // CS0055
   {
      get
      {
         return 0;
      }
   }
}

public class MyClass3
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0058

Article • 09/15/2021

Inconsistent accessibility: return type 'type' is less accessible than delegate 'delegate'

A public construct must return a publicly accessible object. For more information, see Access Modifiers.

The following sample generates CS0058 because no access modifier is applied to MyClass and therefore it is given private accessibility by default:

C#

```
// CS0058.cs
class MyClass
// try the following line instead
// public class MyClass
{
}

public delegate MyClass MyClassDel();   // CS0058

public class A
{
   public static void Main()
   {
   }
}
```

See also

- private

Compiler Error CS0059

Article • 09/15/2021

Inconsistent accessibility: parameter type 'type' is less accessible than delegate 'delegate'

The return type and each of the types referenced in the formal parameter list of a method must be at least as accessible as the method itself. For more information, see Access Modifiers.

Example

The following sample generates CS0059:

C#

```
// CS0059.cs
class MyClass //defaults to private accessibility
// try the following line instead
// public class MyClass
{
}

public delegate void MyClassDel( MyClass myClass);   // CS0059

public class Program
{
    public static void Main()
    {
    }
}
```

Compiler Error CS0060

06/24/2025

Inconsistent accessibility: base class 'Class1' is less accessible than class 'Class2'

In C#, a derived class cannot have broader accessibility than its base class. If the base class is less accessible, consumers of the derived class might unintentionally gain access to a class they shouldn't see.

Subclasses can never be more accessible than their base classes. That would allow consumers of the subclass broader access to the base type than was intended.

The following sample produces CS0060 because `Class1` is `internal` and `Class2` is `public`.

C#

```
internal class Class1
{
}

public class Class2 : Class1 // 🛑 CS0060: Class1 is less accessible
{
}
```

You can resolve CS0060 in one of two ways:

- Make the base class more accessible: Change `Class1` to be `public`.
- Restrict the derived class's accessibility: Change `Class2` to be `internal`.

> **Tip**
>
> Base classes can be *more* accessible than their subclasses, but never *less* accessible.

See also

- Access Modifiers

Compiler Error CS0061

Article • 09/15/2021

Inconsistent accessibility: base interface 'interface 1' is less accessible than interface 'interface 2'

A public construct must return a publicly accessible object.

Interface accessibility cannot be narrowed in a derived interface. For more information, see Interfaces and Access Modifiers.

The following sample generates CS0061.

C#

```
// CS0061.cs
// compile with: /target:library
internal interface A {}
public interface AA : A {}  // CS0061

// OK
public interface B {}
internal interface BB : B {}

internal interface C {}
internal interface CC : C {}
```

Compiler Error CS0065

Article • 09/15/2021

'event' : event property must have both add and remove accessors

An event that is not a field must have both access methods.

The following sample generates CS0065:

C#

```
// CS0065.cs
using System;
public delegate void EventHandler(object sender, int e);
public class MyClass
{
   public event EventHandler Click   // CS0065,
   {
      // to fix, uncomment the add and remove definitions
      /*
      add
      {
         Click += value;
      }
      remove
      {
         Click -= value;
      }
      */
   }

   public static void Main()
   {
   }
}
```

See also

- Events

Compiler Error CS0066

Article • 09/15/2021

'event': event must be of a delegate type

The event keyword requires a delegate type. For more information, see Events and Delegates.

The following sample generates CS0066:

C#

```
// CS0066.cs
using System;

public class EventHandler
{
}

// to fix the error, remove the event declaration and the
// EventHandler class declaration, and uncomment the following line
// public delegate void EventHandler();

public class a
{
   public event EventHandler Click;   // CS0066

   private void TestMethod()
   {
      if (Click != null)
         Click();
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0068

Article • 09/15/2021

'event': event in interface cannot have initializer

An event in an interface cannot have an initializer. For more information, see Interfaces.

The following sample generates CS0068:

C#

```
// CS0068.cs

delegate void MyDelegate();

interface I
{
   event MyDelegate d = new MyDelegate(M.f);   // CS0068
   // try the following line instead
   // event MyDelegate d2;
}

class M
{
   event MyDelegate d = new MyDelegate(M.f);

   public static void f()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0069

Article • 09/15/2021

An event in an interface cannot have add or remove accessors

You cannot define an event's accessor functions in an interface. For more information, see Events and Interfaces.

The following sample generates CS0069:

C#

```
// CS0069.cs
// compile with: /target:library

public delegate void EventHandler();

public interface a
{
   event EventHandler Click { remove {} }   // CS0069
   event EventHandler Click2;   // OK
}
```

Compiler Error CS0070

Article • 09/15/2021

The event 'event' can only appear on the left hand side of += or -= (except when used from within the type 'type')

Outside of the class it is defined in, an event can only add or subtract references. For more information, see Events.

The following sample generates CS0070:

C#

```
// CS0070.cs
using System;
public delegate void EventHandler();

public class A
{
   public event EventHandler Click;

   public static void OnClick()
   {
      EventHandler eh;
      A a = new A();
      eh = a.Click;
   }

   public static void Main()
   {
   }
}

public class B
{
   public int Foo ()
   {
      EventHandler eh = new EventHandler(A.OnClick);
      A a = new A();
      eh = a.Click;   // CS0070
      // try the following line instead
      // a.Click += eh;
      return 1;
   }
}
```

Compiler Error CS0072

Article • 09/15/2021

'event' : cannot override; 'method' is not an event

An event can only override another event. For more information, see Events.

Example

The following sample generates CS0072:

C#

```
// CS0072.cs
delegate void MyDelegate();

class Test1
{
    public virtual event MyDelegate MyEvent;
    public virtual void VMeth()
    {
    }

    public void FireAway()
    {
        if (MyEvent != null)
            MyEvent();
    }
}

class Test2 : Test1
{
    public override event MyDelegate VMeth   // CS0072
    // Uncomment the following lines to resolve.
    // public override event MyDelegate MyEvent
    {
        add
        {
            VMeth += value;
            // MyEvent += value;
        }
        remove
        {
            VMeth -= value;
            // MyEvent -= value;
        }
   }

   public static void Main()
   {
```

```
    }
}
```

Compiler Error CS0073

Article • 09/15/2021

An add or remove accessor must have a body

An **add** or **remove** keyword in an event definition must have a body. For more information, see Events.

The following sample generates CS0073:

C#

```
// CS0073.cs
delegate void del();

class Test
{
   public event del MyEvent
   {
      add;   // CS0073
      // try the following lines instead
      // add
      // {
      //    MyEvent += value;
      // }
      remove
      {
         MyEvent -= value;
      }

   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0074

Article • 09/15/2021

'event': abstract event cannot have initializer

If an event is marked as **abstract**, it cannot be initialized. For more information, see Events.

The following sample generates CS0074:

C#

```
// CS0074.cs
delegate void D();

abstract class Test
{
   public abstract event D e = null;   // CS0074
   // try the following line instead
   // public abstract event D e;

   public static void Main()
   {
   }
}
```

Compiler Error CS0075

Article • 09/15/2021

To cast a negative value, you must enclose the value in parentheses

If you are casting using a keyword that identifies a predefined type, then you do not need parentheses. Otherwise, you must put the parentheses because (x) –y will not be considered a cast expression. From the C# Specification, Section 7.6.6:

From the disambiguation rule it follows that, if x and y are identifiers, (x)y, (x)(y), and (x)(-y) are cast-expressions, but (x)-y is not, even if x identifies a type. However, if x is a keyword that identifies a predefined type (such as int), then all four forms are cast-expressions (because such a keyword could not possibly be an expression by itself).

The following code generates CS0075:

C#

```
// CS0075
namespace MyNamespace
{
    enum MyEnum { }
    public class MyClass
    {
        public static void Main()
        {
            // To fix the error, place the negative
            // values below in parentheses
            int i = (System.Int32) - 4; //CS0075
            MyEnum e = (MyEnum) - 1;    //CS0075
            System.Console.WriteLine(i); //to avoid warning
            System.Console.WriteLine(e); //to avoid warning
        }
    }
}
```

See also

- Casting and Type Conversions

Compiler Error CS0076

Article • 10/11/2023

The enumerator name 'value__' is reserved and cannot be used

An item in an enumeration cannot have an identifier called **value__** since it's reserved for use as the enumeration's backing field.

The following sample generates CS0076:

C#

```
// CS0076.cs
public enum E
{
    value__   // CS0076
}
```

Compiler Error CS0077

Article • 10/18/2024

The as operator must be used with a reference type or nullable type ('int' is a non-nullable value type).

The as operator was passed a value type. Because `as` can return null, it can only be passed a reference type or a nullable value type.

However, using pattern matching with the is operator, we can directly perform type checking and assignments in one step.

The following sample generates CS0077:

C#

```
// CS0077.cs
using System;

struct S
{
}

class M
{
   public static void Main()
   {
      object o;
      S s;

      o = new S();

      s = o as S;    // CS0077, S is not a reference type

      // Use pattern matching instead of as
      if (o is S sValue)
      {
          s = sValue;
      }
   }
}
```

Compiler Error CS0079

Article • 09/15/2021

The event 'event' can only appear on the left hand side of += or -=

An event was called incorrectly. For more information, see Events and Delegates.

The following sample generates CS0079:

C#

```
// CS0079.cs
using System;

public delegate void MyEventHandler();

public class Class1
{
   private MyEventHandler _e;

   public event MyEventHandler Pow
   {
      add
      {
         _e += value;
         Console.WriteLine("in add accessor");
      }
      remove
      {
         _e -= value;
         Console.WriteLine("in remove accessor");
      }
   }

   public void Handler()
   {
   }

   public void Fire()
   {
      if (_e != null)
      {
         Pow();   // CS0079
         // try the following line instead
         // _e();
      }
   }

   public static void Main()
   {
      Class1 p = new Class1();
```

```
        p.Pow += new MyEventHandler(p.Handler);
        p._e();
        p.Pow += new MyEventHandler(p.Handler);
        p._e();
        p._e -= new MyEventHandler(p.Handler);
        if (p._e != null)
        {
            p._e();
        }
        p.Pow -= new MyEventHandler(p.Handler);
        if (p._e != null)
        {
            p._e();
        }
    }
}
```

Compiler Error CS0082

Article • 09/15/2021

Type 'type' already reserves a member called 'name' with the same parameter types

Properties at compile time are translated to methods with `get_` and/or `set_` in front of the identifier. If you define your own method that conflicts with the method name, an error is generated.

Example

The following example generates CS0082:

C#

```
//cs0082.cs
class MyClass
{

    //property
    public int MyProp
    {
        get //CS0082
        {
            return 1;
        }
    }

    //conflicting Get
    public int get_MyProp()
    {
        return 2;
    }

    public static int Main()
    {
        return 1;
    }
}
```

See also

- Properties

Compiler Error CS0100

Article • 09/15/2021

The parameter name 'parameter name' is a duplicate

A method declaration used the same parameter name more than once. Parameter names must be unique in a method declaration. For more information, see Methods.

The following sample generates CS0100:

C#

```
// CS0100.cs
namespace x
{
   public class a
   {
      public static void f(int i, char i)   // CS0100
      // try the following line instead
      // public static void f(int i, char j)
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0101

Article • 09/15/2021

The namespace 'namespace' already contains a definition for 'type'

A namespace has duplicate identifiers. Rename or delete one of the duplicate identifiers. For more information, see Namespaces

The following sample generates CS0101:

C#

```
// CS0101.cs
namespace MyNamespace
{
   public class MyClass
   {
      static public void Main()
      {
      }
   }

   public class MyClass   // CS0101
   {
   }
}
```

A CS0101 is also generated when your class name clashes with your namespace name. This can happen when expanding with helper classes for the base class where you attempt to keep the namespace route the same. In the below example, the UTF8 class should clearly be a subsidiary of the String class, but attempting to force it into the same name space by declaring said namespace as Utilities.String will cause a CS0101 error:

C#

```
//CS0101-Utilities.String.cs
namespace Utilities
{
   public class String
   {
        public string MyString;
   }
}

//CS0101-Utilities.String.UTF8.cs
namespace Utilities.String  // CS0101
{
   public class UTF8
```

```
    {
        public string MySecondString;
    }
}
```

Compiler Error CS0102

Article • 09/15/2021

The type 'type name' already contains a definition for 'identifier'

A class contains multiple declarations for identifiers with the same name at the same scope. To fix the error, rename the duplicate identifiers.

Example

The following sample generates CS0102.

C#

```
// CS0102.cs
// compile with: /target:library
namespace MyApp
{
   public class MyClass
   {
      string s = "Hello";
      string s = "Goodbye";   // CS0102

      public void GetString()
      {
         string s = "Hello again";   // method scope, no error
      }
   }
}
```

Compiler Error CS0103

Article • 10/26/2022

The name 'identifier' does not exist in the current context

An attempt was made to use a name that does not exist in the class, namespace, or scope. Check the spelling of the name and check your `using` directives and assembly references to make sure that the name that you are trying to use is available.

This error frequently occurs if you declare a variable in a loop or a `try` or `if` block and then attempt to access it from an enclosing code block or a separate code block, as shown in the following example:

> **Note**
>
> This error may also be presented when missing the `greater than` symbol in the operator `=>` in an expression lambda. For more information, see expression lambdas.

C#

```
using System;

class MyClass1
{
    public static void Main()
    {
        try
        {
            // The following declaration is only available inside the try
block.
            var conn = new MyClass1();
        }
        catch (Exception e)
        {
            // The following expression causes error CS0103, because
variable
            // conn only exists in the try block.
            if (conn != null)
                Console.WriteLine("{0}", e);
        }
    }
}
```

The following example resolves the error:

C#

```
using System;

class MyClass2
{
    public static void Main()
    {
        // To resolve the error in the example, the first step is to
        // move the declaration of conn out of the try block. The following
        // declaration is available throughout the Main method.
        MyClass2 conn = null;
        try
        {
            // Inside the try block, use the conn variable that you declared
            // previously.
            conn = new MyClass2();
        }
        catch (Exception e)
        {
            // The following expression no longer causes an error, because
            // the declaration of conn is in scope.
            if (conn != null)
                Console.WriteLine("{0}", e);
        }
    }
}
```

Compiler Error CS0104

Article • 10/06/2022

'reference' is an ambiguous reference between 'identifier' and 'identifier'

Your program contains using directives for two namespaces and your code references a name that appears in both namespaces.

The following sample generates CS0104:

C#

```
// CS0104.cs
using x;
using y;

namespace x
{
   public class Test
   {
   }
}

namespace y
{
   public class Test
   {
   }
}

public class a
{
   public static void Main()
   {
      Test test = new Test();   // CS0104, is Test in x or y namespace?
      // try the following line instead
      // y.Test test = new y.Test();
   }
}
```

Compiler Error CS0107

Article • 09/15/2021

More than one protection modifier

Only one access modifier (public, private, protected, or internal) is allowed on a class member, with the exception of **protected internal** and **private protected**.

The following sample generates CS0107:

C#

```
// CS0107.cs
public class C
{
    private internal void F()   // CS0107, delete private or internal
    {
    }
}
```

Compiler Error CS0110

Article • 02/29/2024

The evaluation of the constant value for 'const declaration' involves a circular definition

The declaration of a const variable cannot reference another const variable such that a circular dependency is formed. This also applies to associated constant values of enum members.

Example

The associated constant value of `MyClass.Color.Red` is explicitly set to `MyClass.Color.Blue`, but the value of `MyClass.Color.Blue` is dependent on the previous enum member (`MyClass.Color.Red`), hence both constant values cannot be determined. Similarly, the constant variable `MyClass.a` is defined in terms of `MyClass.b`, but that is also defined in terms of `MyClass.a`.

The following sample generates CS0110:

C#

```
// CS0110.cs
// compile with: /target:library
class MyClass
{
    enum Color
    {
        Red = Blue,   // CS0110
        Blue,
    }

    public const int a = b + 1;   // CS0110
    public const int b = a + 1;
}
```

To resolve this error, break the circular reference by modifying or removing the definition.

See also

- Constants

Compiler Error CS0112

Article • 09/15/2021

A static member 'function' cannot be marked as override, virtual or abstract

Any method declaration that uses the `override`, **virtual**, or **abstract** keyword cannot also use the **static** keyword.

For more information, see Methods.

The following sample generates CS0112:

C#

```
// CS0112.cs
namespace MyNamespace
{
   abstract public class MyClass
   {
      public abstract void Foo();
   }
   public class MyClass2 : MyClass
   {
      override public static void Foo()   // CS0112, remove static keyword
      {
      }
      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0113

Article • 09/15/2021

A member 'function' marked as override cannot be marked as new or virtual

It is mutually exclusive to mark a method with the new and override keywords.

The following sample generates CS0113:

C#

```
// CS0113.cs
namespace MyNamespace
{
   abstract public class MyClass
   {
      public abstract void Foo();
   }

   public class MyClass2 : MyClass
   {
      override new public void Foo()   // CS0113, remove new keyword
      {
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0115

Article • 09/15/2021

'function' : no suitable method found to override

A method was marked as an override, but the compiler found no method to override. For more information, see override, and Knowing When to Use Override and New Keywords.

Example

The following sample generates CS0115. You can resolve CS0115 in one of two ways:

- Remove the `override` keyword from the method in `MyClass2`.
- Use `MyClass1` as a base class for `MyClass2`.

C#

```
// CS0115.cs
namespace MyNamespace
{
    abstract public class MyClass1
    {
        public abstract int f();
    }

    abstract public class MyClass2
    {
        public override int f()   // CS0115
        {
            return 0;
        }

        public static void Main()
        {
        }
    }
}
```

Compiler Error CS0116

Article • 10/27/2021

A namespace cannot directly contain members such as fields or methods.

A namespace can contain other namespaces, structs, and classes. For more information, see the namespace keyword article.

Example

The following sample will cause Visual Studio to flag parts of the code as being in violation of CS0116. Attempting to build this code will result in build failure:

C#

```
// CS0116.cs
namespace x
{
    // A namespace can be placed within another namespace.
    using System;

    // These variables trigger the CS0116 error as they are declared outside
of a struct or class.
    public int latitude;
    public int longitude;
    Coordinate coord;

    // Auto-properties also fall under the definition of this rule.
    public string LocationName { get; set; }

    // This method as well: if it isn't in a class or a struct, it's
violating CS0116.
    public void DisplayLatitude()
    {
        Console.WriteLine($"Lat: {latitude}");
    }

    public struct Coordinate
    {
    }

    public class CoordinatePrinter
    {
        public void DisplayLongitude()
        {
            Console.WriteLine($"Longitude: {longitude}");
        }

        public void DisplayLocation()
```

```
        {
            Console.WriteLine($"Location: {LocationName}");
        }
    }
}
```

Note that in C#, methods and variables must be declared and defined within a struct or class. For more information on program structure in C#, see the General Structure of a C# Program article. To fix this error, rewrite your code such that all methods and fields are contained within either a struct or a class:

C#

```
namespace x
{
    // A namespace can be placed within another namespace.
    using System;

    // These variables are now placed within a struct, so CS0116 is no
longer violated.
    public struct Coordinate
    {
        public int Latitude;
        public int Longitude;
    }

    // The methods and fields are now placed within a class, and the
compiler is satisfied.
    public class CoordinatePrinter
    {
        Coordinate coord;
        public string LocationName { get; set; }

        public void DisplayLatitude()
        {
            Console.WriteLine($"Lat: {coord.Latitude}");
        }

        public void DisplayLongitude()
        {
            Console.WriteLine($"Longitude: {coord.Longitude}");
        }

        public void DisplayLocation()
        {
            Console.WriteLine($"Location: {LocationName}");
        }
    }
}
```

See also

- General Structure of a C# Program
- The C# type system
- Namespaces

Compiler Error CS0117

Article • 09/15/2021

'type' does not contain a definition for 'identifier'

- This error occurs in some situations when a reference is made to a member that does not exist for the data type.

Example

The following sample generates CS0117:

C#

```
public class Base { }

public class Derived : Base
{
    public void TestInt()
    {
        int i = base.someMember; // CS0117
    }
}
```

Compiler Error CS0118

Article • 09/15/2021

'construct1_name' is a 'construct1' but is used like a 'construct2'

The compiler detected a situation in which a construct was used in some erroneous way or a disallowed operation was tried on a construct. Some common examples include the following:

- A try to instantiate a namespace (instead of a class)
- A try to call a field (instead of a method)
- A try to use a type as a variable
- A try to use an extern alias as a type.

To resolve this error, make sure that the operation you are performing is valid for the type you are performing the operation on.

Example

The following sample generates CS0118.

C#

```
// CS0118.cs
// compile with: /target:library
namespace MyNamespace
{
   class MyClass
   {
      // MyNamespace not a class
      MyNamespace ix = new MyNamespace ();   // CS0118
   }
}
```

Compiler Error CS0119

07/16/2025

'construct1_name' is a 'construct1', which is not valid in the given context.

The compiler detected an unexpected construct such as the following:

- A class constructor is not a valid test expression in a conditional statement.
- A class name was used instead of an instance name to refer to an array element.
- A method identifier is used as if it were a struct or class

Examples

Example 1

The following sample generates CS0119: 'C.B()' is a method, which is not valid in the given context. You can fix this error by changing the name of the method `C.B`, or using the fully qualified name for the class `B` like `N2.B`.

C#

```
namespace N2
{
    public static class B
    {
        public static void X() {}
    }
}

namespace N1
{
    public class C
    {
        void B() {}
        void M() => B.X();   // CS0119
    }
}
```

Example 2

The following sample shows a common scenario where CS0119 occurs when trying to add class names to a collection of types. The compiler expects Type instances, not class names.

C#

```
using System;
using System.Collections.Generic;

public class Page_BaseClass { }
public class Page_CRTable { }
public class Page_DBGridWithTools { }

public static class PageTypeManager
{
    public static List<Type> GetPageTypes()
    {
        List<Type> pageTypesList = new List<Type>();

        // CS0119: 'Page_BaseClass' is a type, which is not valid in the given
context
        pageTypesList.Add(Page_BaseClass);
        pageTypesList.Add(Page_CRTable);
        pageTypesList.Add(Page_DBGridWithTools);

        return pageTypesList;
    }
}
```

To fix this error, use the Type.GetType method or the `typeof` operator to get `Type` instances:

C#

```
using System;
using System.Collections.Generic;

public static class PageTypeManager
{
    public static List<Type> GetPageTypes()
    {
        List<Type> pageTypesList = new List<Type>();

        // Use typeof operator to get Type instances
        pageTypesList.Add(typeof(Page_BaseClass));
        pageTypesList.Add(typeof(Page_CRTable));
        pageTypesList.Add(typeof(Page_DBGridWithTools));

        return pageTypesList;
    }

    // Alternative: Load types dynamically by name
    public static List<Type> GetPageTypesByName(string[] typeNames)
    {
        List<Type> pageTypesList = new List<Type>();

        foreach (string typeName in typeNames)
        {
```

```
            Type type = Type.GetType(typeName);
            if (type != null)
            {
                pageTypesList.Add(type);
            }
        }

        return pageTypesList;
    }
}
```

Compiler Error CS0120

Article • 02/24/2023

An object reference is required for the non-static field, method, or property 'member'

In order to use a non-static field, method, or property, you must first create an object instance. For more information about static methods, see Static Classes and Static Class Members. For more information about creating instances of classes, see Instance Constructors.

Example 1

The following sample generates CS0120:

C#

```
// CS0120_1.cs
public class MyClass
{
    // Non-static field.
    public int i;
    // Non-static method.
    public void Method() { }
    // Non-static property.
    public int Prop
    {
        get
        {
            return 1;
        }
    }

    public static void Main()
    {
        i = 10;   // CS0120
        Method();   // CS0120
        int p = Prop;   // CS0120
    }
}
```

To correct this error, first create an instance of the class:

C#

```
// CS0120_1.cs
public class MyClass
{
```

```
    // Non-static field.
    public int i;
    // Non-static method.
    public void Method() { }
    // Non-static property.
    public int Prop
    {
        get
        {
            return 1;
        }
    }

    public static void Main()
    {
        var mc = new MyClass();
        mc.i = 10;
        mc.Method();
        int p = mc.Prop;
    }
}
```

Example 2

CS0120 will also be generated if there is a call to a non-static method from a static method, as follows:

C#

```
// CS0120_2.cs
public class MyClass
{
    public void Method() { }

    public static void Main()
    {
        Method();   // CS0120
    }
}
```

To correct this error, first create an instance of the class:

C#

```
// CS0120_2.cs
public class MyClass
{
    public void Method() { }

    public static void Main()
```

```
    {
        var mc = new MyClass();
        mc.Method();
    }
}
```

Example 3

Similarly, a static method cannot call an instance method unless you explicitly give it an instance of the class, as follows:

C#

```
// CS0120_3.cs
using System;

public class MyClass
{
    public static void Main()
    {
        DoIt("Hello There");   // CS0120
    }

    private void DoIt(string sText)
    {
        Console.WriteLine(sText);
    }
}
```

To correct this error, you could also add the keyword static to the method definition:

C#

```
// CS0120_3.cs
using System;

public class MyClass
{
    public static void Main()
    {
        DoIt("Hello There");
    }

    private static void DoIt(string sText)
    {
        Console.WriteLine(sText);
    }
}
```

See also

- The C# type system

Compiler Error CS0122

Article • 10/05/2023

'member' is inaccessible due to its protection level

The access modifier for a class member prevents accessing the member. For more information, see Access Modifiers.

Extension methods cannot access private members of the type they are extending.

One cause of this (not shown in the sample below) can be omitting the **/out** compiler flag on the target of a friend assembly. For more information, see Friend Assemblies and **OutputAssembly** (C# Compiler Options).

Example

The following sample generates CS0122:

C#

```
// CS0122.cs
public class MyClass
{
    private int data;

    void PrivateMethod() {}
    public void PublicMethod() {}
}

public static class MyClassExtensions
{
    public static int GetData(this MyClass myClass)
    {
        return myClass.data;   // CS0122
    }
}

public class Program
{
    public static void Main()
    {
        MyClass a = new MyClass();
        a.PrivateMethod();   // CS0122
        a.PublicMethod();   // OK
    }
}
```

Compiler Error CS0123

Article • 09/15/2021

No overload for 'method' matches delegate 'delegate'

An attempt to create a delegate failed because the correct signature was not used. Instances of a delegate must be declared with the same signature as the delegate declaration.

You can resolve this error by adjusting either the method or delegate signature. For more information, see Delegates.

The following sample generates CS0123.

C#

```
// CS0123.cs
delegate void D();
delegate void D2(int i);

public class C
{
   public static void f(int i) {}

   public static void Main()
   {
      D d = new D(f);   // CS0123
      D2 d2 = new D2(f);   // OK
   }
}
```

Compiler Error CS0126

Article • 09/15/2021

An object of a type convertible to 'type' is required

A return statement is present, but the statement does not return a value of the expected type. For more information, see Properties.

The following sample generates CS0126:

C#

```
// CS0126.cs
public class a
{
   public int i
   {
      set
      {
      }
      get
      {
         return;   // CS0126, specify a value to return
      }
   }
}
```

Compiler Error CS0127

Article • 09/15/2021

Since 'function' returns void, a return keyword must not be followed by an object expression

A method with a void return type cannot return a value. For more information, see Methods.

The following sample generates CS0127:

C#

```
// CS0127.cs
namespace MyNamespace
{
   public class MyClass
   {
      public int hiddenMember2
      {
         get
         {
            return 0;
         }
         set   // CS0127, set has an implicit void return type
         {
            return 0;   // remove return statement to resolve this CS0127
         }
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0128

Article • 09/15/2021

A local variable named 'variable' is already defined in this scope

The compiler detected declarations of two local variables with the same name. For more information, see Methods.

The following sample generates CS0128:

C#

```
// CS0128.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         char i;
         int i;   // CS0128
      }
   }
}
```

Compiler Error CS0131

09/06/2025

The left-hand side of an assignment must be a variable, property or indexer

In an assignment statement, the value of the right-hand side is assigned to the left-hand side. The left-hand side must be a variable, property, or indexer.

To fix this error, make sure that all operators are on the right-hand side and that the left-hand side is a variable, property, or indexer. For more information, see Operators and expressions.

Example 1

The following sample generates CS0131.

C#

```
// CS0131.cs
public class MyClass
{
    public int i = 0;
    public void MyMethod()
    {
        i++ = 1;   // CS0131
        // try the following line instead
        // i = 1;
    }
    public static void Main() { }
}
```

Example 2

The following sample generates CS0131 when assigning to a constant field.

C#

```
// CS0131b.cs
public class B
{
    public static int Main()
    {
        const int j = 0;
        j = 1; // CS0131
        // try the following lines instead
        // int j = 0;
        // j = 1;
```

```
        return j;
    }
}
```

Example 3

This error can also occur if you attempt to perform arithmetic operations on the left hand side of an assignment operator, as in the following example.

C#

```
// CS0131c.cs
public class C
{
    public static int Main()
    {
        int a = 1, b = 2, c = 3;
        if (a + b = c) // CS0131
        // try this instead
        // if (a + b == c)
            return 0;
        return 1;
    }
}
```

Compiler Error CS0132

10/06/2025

'constructor' : a static constructor must be parameterless

A static constructor cannot be declared with one or more parameters. For more information, see Static Constructors.

The following sample generates CS0132:

C#

```
// CS0132.cs
namespace MyNamespace
{
   public class MyClass
   {
      public MyClass(int i)
      {
      }
   }

   public class MyClass2 : MyClass
   {
      static MyClass2(int i)   // CS0132
      {
      }
   }
}
```

Compiler Error CS0133

Article • 09/15/2021

The expression being assigned to 'variable' must be constant

A const variable cannot take as its value an expression that is not constant. For more information, see Constants.

The following sample generates CS0133:

C#

```
// CS0133.cs
public class MyClass
{
   public const int i = c;   // CS0133, c is not constant
   public static int c = i;
   // try the following line instead
   // public const int i = 6;

   public static void Main()
   {
   }
}
```

Compiler Error CS0134

Article • 09/15/2021

'variable' is of type 'type'. A const field of a reference type other than string can only be initialized with null.

A constant-expression is an expression that can be fully evaluated at compile-time. Because the only way to create a non-null value of a reference-type is to apply the new operator, and because the new operator is not permitted in a constant-expression, the only possible value for constants of reference-types other than string is null.

If you encounter this error by trying to create a const string array, the solution is to make the array readonly, and initialize it in the constructor.

Example

The following example generates CS0134:

C#

```
// CS0134.cs
// compile with: /target:library
class MyTest {}

class MyClass
{
    const MyTest test = new MyTest();   // CS0134

    //OK
    const MyTest test2 = null;
    const System.String test3 = "test";
}
```

Compiler Error CS0135

Article • 01/25/2022

'declaration1' conflicts with the declaration 'declaration2'

The compiler does not allow hiding names, which commonly leads to logic errors in your code.

Example

The following sample generates CS0135:

C#

```
// CS0135.cs
public class MyClass2
{
   public static int i = 0;

   public static void Main()
   {
      {
         int i = 4;
         i++;
      }
      i = 0;   // CS0135
   }
}
```

From the C# Language Specification:

It is an error for a local variable declaration space and a nested local variable declaration space to contain elements with the same name. Thus, within a nested declaration space it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.

Compiler Error CS0136

Article • 05/08/2025

A local variable named 'var' cannot be declared in this scope because it would give a different meaning to 'var', which is already used in a 'parent or current/child' scope to denote something else

A variable declaration hides another declaration that would otherwise be in scope. Rename the variable that is declared on the line that generated CS0136.

Example

The following sample generates CS0136:

C#

```
// CS0136.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         int i = 0;
         {
            char i = 'a';   // CS0136, hides int i
         }
         i++;
      }
   }
}
```

The compiler reports this error regardless of the textual order of the variable declarations, as shown in the following example:

C#

```
// CS0136.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         if (true)
         {
            int i = 1;   // CS0136, hides i outside this block
         }
```

```
            int i = 0;
            i++;
        }
    }
}
```

From the C# Language Specification:

It is an error for a local variable declaration space and a nested local variable declaration space to contain elements with the same name. Thus, within a nested declaration space it is not possible to declare a local variable or constant with the same name as a local variable or constant in an enclosing declaration space. It is possible for two declaration spaces to contain elements with the same name as long as neither declaration space contains the other.

Compiler Error CS0139

Article • 12/09/2021

No enclosing loop out of which to break or continue

A break or continue statement was encountered outside of a loop.

For more information, see the break and continue statements.

The following sample generates CS0139 twice:

C#

```
// CS0139.cs
namespace x
{
   public class a
   {
      public static void Main()
      {
         continue;  // CS0139
         break;     // CS0139
      }
   }
}
```

Compiler Error CS0140

Article • 12/09/2021

The label 'label' is a duplicate

A label with the same name appeared twice. For more information, see goto.

The following sample generates CS0140:

C#

```
// CS0140.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         label1: int i = 0;
         label1: int j = 0;   // CS0140, comment this line to resolve
         goto label1;
      }
   }
}
```

Compiler Error CS0143

Article • 09/15/2021

The type 'class' has no constructors defined

There is no appropriate constructor available. This is the case for built-in numeric value types, which are initialized simply by assigning a value to them.

The following sample generates CS0143:

C#

```
// CS0143.cs
class MyClass
{
   static public void Main ()
   {
      double d = new double(4.5);   // CS0143
      // Try this line instead:
      // double d = 4.5;
   }
}
```

Compiler Error CS0144

Article • 03/18/2022

Cannot create an instance of the abstract class or interface 'interface'

You cannot create an instance of an abstract class or an interface. For more information, see Interfaces.

The following sample generates CS0144:

```
C#
```

```
// CS0144.cs
interface MyInterface
{
}
public class MyClass
{
   public static void Main()
   {
      MyInterface myInterface = new MyInterface ();   // CS0144
   }
}
```

How to fix violations

You can solve this problem by implementing one of the two following solutions:

1. Change the type declaration so that it's not abstract: Either remove the abstract keyword from the class declaration, or change the type from an interface to a class.
2. Create a type that's derived from the abstract class or that implements the interface.

Compiler Error CS0145

Article • 09/15/2021

A const field requires a value to be provided

You must initialize a const variable. For more information, see Constants.

The following sample generates CS0145:

C#

```
// CS0145.cs
class MyClass
{
   const int i;   // CS0145
   // try the following line instead
   // const int i = 0;

   public static void Main()
   {
   }
}
```

Compiler Error CS0146

Article • 09/15/2021

Circular base class dependency involving 'class1' and 'class2'

The inheritance list for a class includes a direct or indirect reference to itself. A class cannot inherit from itself. For more information, see Inheritance.

The following sample generates CS0146:

C#

```
// CS0146.cs
namespace MyNamespace
{
   public interface InterfaceA
   {
   }

   public class MyClass : InterfaceA, MyClass2
   {
      public void Main()
      {
      }
   }

   public class MyClass2 : MyClass   // CS0146
   {
   }
}
```

Compiler Error CS0148

07/22/2025

The delegate 'delegate' does not have a valid constructor

You imported and used a managed program (one that uses the .NET runtime) that was created with another compiler. That compiler allowed an ill-formed delegate constructor. For more information, see Delegates.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

Compiler Error CS0149

Article • 09/15/2021

Method name expected

When creating a delegate, specify a method. For more information, see Delegates.

The following sample generates CS0149:

C#

```
// CS0149.cs
using System;

delegate string MyDelegate(int i);

class MyClass
{
   // class member-field of the declared delegate type
   static MyDelegate dt;

   public static void Main()
   {
      dt = new MyDelegate(17.45);   // CS0149
      // try the following line instead
      // dt = new MyDelegate(Func2);
      F(dt);
   }

   public static string Func2(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void F(MyDelegate myFunc)
   {
      myFunc(8);
   }
}
```

Compiler Error CS0150

Article • 07/30/2022

A constant value is expected

A variable was found where a constant was expected. For more information, see switch.

The following sample generates CS0150:

C#

```
// CS0150.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         int i = 0;
         int j = 0;

         switch(i)
         {
            case j:   // CS0150, j is a variable int, not a constant int
            // try the following line instead
            // case 0:
         }
      }
   }
}
```

This error is also produced when an array size is specified with a variable value and initialized with an array initializer. To remove the error, initialize the array in a separate statement or statements.

C#

```
// CS0150.cs
    namespace MyNamespace
    {
        public class MyClass
        {
            public static void Main()
            {
                int size = 2;
                double[] nums = new double[size] { 46.9, 89.4 }; //CS0150
                // Try the following lines instead
                // double[] nums = new double[size];
                // nums[0] = 46.9;
```

```
                // nums[1] = 89.4;
            }
        }

    }
```

Compiler Error CS0151

Article • 10/27/2021

A value of an integral type expected

A variable was used in a situation where an integral data type was required. For more information, see Types.

Example of ambiguous conversion

This error can occur when there is no conversion or if the available implicit conversions result in an ambiguous situation. The following example generates CS0151:

C#

```
public class MyClass
{
   public static implicit operator int (MyClass aa)
   {
      return 0;
   }

   public static implicit operator long (MyClass aa)
   {
      return 0;
   }

   public static void Main()
   {
      MyClass a = new MyClass();

      // Compiler cannot choose between int and long.
      switch (a)   // CS0151
      // try the following line instead
      // switch ((int)a)
      {
         case 1:
            break;
      }
   }
}
```

Compiler Error CS0152

Article • 09/15/2021

The label 'label' already occurs in this switch statement

A label was repeated in a switch statement.

The following sample generates CS0152:

C#

```
// CS0152.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         int i = 0;

         switch (i)
         {
            case 1:
               i++;
               return;

            case 1:   // CS0152, two case 1 statements
               i++;
               return;
         }
      }
   }
}
```

Compiler Error CS0153

Article • 09/15/2021

A goto case is only valid inside a switch statement

An attempt was made to use the switch statement syntax outside of a `switch` statement.

The following sample generates CS0153:

C#

```
// CS0153.cs
public class a
{
   public static void Main()
   {
      goto case 5;   // CS0153
   }
}
```

Compiler Error CS0154

Article • 09/15/2021

The property or indexer 'property' cannot be used in this context because it lacks the get accessor

An attempt to use a property failed because no get accessor method was defined in the property. For more information, see Fields.

Example

The following sample generates CS0154:

C#

```
// CS0154.cs
public class MyClass2
{
    public int i
    {
        set
        {
        }
        // uncomment the get method to resolve this error
        /*
        get
        {
            return 0;
        }
        */
    }
}

public class MyClass
{
    public static void Main()
    {
        MyClass2 myClass2 = new MyClass2();
        int j = myClass2.i;   // CS0154, no get method
    }
}
```

Compiler Error CS0155

Article • 04/22/2023

The type caught or thrown must be derived from System.Exception

An attempt was made to pass a data type that does not derive from **System.Exception** into a catch block. Only data types that derive from **System.Exception** can be passed into a **catch** block. For more information, see Exceptions and Exception Handling.

The following sample generates CS0155:

C#

```
// CS0155.cs
using System;

namespace MyNamespace
{
    public class MyClass2
    // try the following line instead
    // public class MyClass2 : Exception
    {
    }
    public class MyClass
    {
        public static void Main()
        {
            try
            {
            }
            catch (MyClass2)   // CS0155, resolves if you derive MyClass2
from Exception
            {
            }
        }
    }
}
```

Compiler Error CS0156

Article • 04/19/2023

A throw statement with no arguments is not allowed outside of a catch clause

A throw statement with no parameters can only appear in a **catch** clause that takes no parameters.

For more information, see Exceptions and Exception Handling.

The following sample generates CS0156:

C#

```
// CS0156.cs
using System;

namespace MyNamespace
{
   public class MyClass2 : Exception
   {
   }

   public class MyClass
   {
      public static void Main()
      {
         try
         {
            throw;   // CS0156
         }

         catch(MyClass2)
         {
            throw;   // this throw is valid
         }
      }
   }
}
```

Compiler Error CS0157

Article • 04/22/2023

Control cannot leave the body of a finally clause

All of the statements in a finally clause must execute. For more information, see Exceptions and Exception Handling.

The following sample generates CS0157:

C#

```
// CS0157.cs
using System;
namespace MyNamespace
{
   public class MyClass2 : Exception
   {
   }

   public class MyClass
   {
      public static void Main()
      {
         try
         {
         }
         finally
         {
            return;   // CS0157, cannot leave finally clause
         }
      }
   }
}
```

Compiler Error CS0158

Article • 12/09/2021

The label 'label' shadows another label by the same name in a contained scope

A label in an inner scope hides a label with the same name in an outer scope. For more information, see goto.

The following sample generates CS0158:

C#

```
// CS0158.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         goto lab1;
         lab1:
         {
            lab1:
            goto lab1;   // CS0158
         }
      }
   }
}
```

Compiler Error CS0159

Article • 12/09/2021

No such label 'label' within the scope of the goto statement

The label referenced by the goto statement could not be found within the scope of the `goto` statement.

The following sample generates CS0159:

C#

```
// CS0159.cs
public class Class1
{
   public static void Main()
   {
      int i = 0;

      switch (i)
      {
         case 1:
            goto case 3;   // CS0159, case 3 label does not exist
         case 2:
            break;
      }
      goto NOWHERE;   // CS0159, NOWHERE label does not exist
   }
}
```

Compiler Error CS0160

Article • 04/22/2023

A previous catch clause already catches all exceptions of this or of a super type ('type')

A series of catch clauses needs to be in decreasing order of derivation. For example, the most derived objects must appear first.

For more information, see Exceptions and Exception Handling.

The following sample generates CS0160:

C#

```
// CS0160.cs
public class MyClass2 : System.Exception {}
public class MyClass
{
   public static void Main()
   {
      try {}

      catch(System.Exception) {}   // Second-most derived; should be second
catch
      catch(MyClass2) {}   // CS0160  Most derived; should be first catch
   }
}
```

Compiler Error CS0161

Article • 09/15/2021

'method': not all code paths return a value

A method that returns a value must have a `return` statement in all code paths. For more information, see Methods.

Example

The following sample generates CS0161:

C#

```
// CS0161.cs
public class Test
{
    public static int Main() // CS0161
    {
        int i = 5;
        if (i < 10)
        {
            return i;
        }
        else
        {
            // Uncomment the following line to resolve.
            // return 1;
        }
    }
}
```

Compiler Error CS0163

10/09/2025

Control cannot fall through from one case label ('label') to another

When a switch statement contains more than one switch section, you must explicitly terminate each section, including the last one, by using one of the following keywords:

- return
- goto
- break
- throw

If you want to implement "fall through" behavior from one section to the next, use `goto case #`.

The following sample generates CS0163.

C#

```
// CS0163.cs
public class MyClass
{
    public static void Main()
    {
        int i = 0;

        switch (i)   // CS0163
        {
            // Compiler error CS0163 is reported on the following line.
            case 1:
                i++;
            // To resolve the error, uncomment one of the following example
statements.
            // return;
            // break;
            // goto case 3;

            case 2:
                i++;
                return;

            case 3:
                i = 0;
                break;

            default:
                Console.WriteLine("Default");
                break;
```

```
        }
    }
}
```

Note that it is correct to have several `case`s for one implementation, like in the following snippet:

C#

```
public class MyClass
{
    public static void Main()
    {
        int i = 0;

        switch(i)
        {
            case 1:
            case 2:    // No CS0163
                i++
                break;

            default:
                break;
        }
    }
}
```

Compiler Error CS0165

Article • 09/15/2021

Use of unassigned local variable 'name'

The C# compiler doesn't allow the use of uninitialized variables. If the compiler detects the use of a variable that might not have been initialized, it generates compiler error CS0165. For more information, see Fields. This error is generated when the compiler encounters a construct that might result in the use of an unassigned variable, even if your particular code does not. This avoids the necessity of overly complex rules for definite assignment.

For more information, see Why does a recursive lambda cause a definite assignment error?.

Example 1

The following sample generates CS0165:

C#

```
// CS0165.cs
using System;

class MyClass
{
    public int i;
}

class MyClass2
{
    public static void Main(string[] args)
    {
        // i and j are not initialized.
        int i, j;

        // You can provide a value for args[0] in the 'Command line
arguments'
        // text box on the Debug tab of the project Properties window.
        if (args[0] == "test")
        {
            i = 0;
        }
        // If the following else clause is absent, i might not be
        // initialized.
        //else
        //{
```

```
        //    i = 1;
        //}

        // Because i might not have been initialized, the following
        // line causes CS0165.
        j = i;

        // To resolve the error, uncomment the else clause of the previous
        // if statement, or initialize i when you declare it.

        // The following example causes CS0165 because myInstance is
        // declared but not instantiated.
        MyClass myInstance;
        // The following line causes the error.
        myInstance.i = 0;

        // To resolve the error, replace the previous declaration with
        // the following line.
        //MyClass myInstance = new MyClass();
    }
}
```

Example 2

Compiler error CS0165 can occur in recursive delegate definitions. You can avoid the error by defining the delegate in two statements so that the variable is not used before it is initialized. The following example demonstrates the error and the resolution.

C#

```
class Program
{
    delegate void Del();
    static void Main(string[] args)
    {
        // The following line causes CS0165 because variable d is used
        // as an argument before it has been initialized.
        Del d = delegate() { System.Console.WriteLine(d); };

        //// To resolve the error, initialize d in a separate statement.
        //Del d = null;
        //// After d is initialized, you can use it as an argument.
        //d = delegate() { System.Console.WriteLine(d); };
        //d();
    }
}
```

Load(String)

Compiler Error CS0167

Article • 09/15/2021

The delegate 'delegate' is missing the Invoke method

You imported and used a managed program (one that uses the .NET runtime) that was created with another compiler. That compiler allowed an ill-formed delegate. Therefore, the `Invoke` method was not available. For more information, see Delegates.

Compiler Error CS0170

Article • 09/15/2021

Use of possibly unassigned field 'field'

A field in a structure was used without first being initialized. To solve this problem, first determine which field was uninitialized and then initialize it before you try to access it. For more information about initializing structs, see Structure types.

The following sample generates CS0170:

C#

```
// CS0170.cs
public struct error
{
   public int i;
}

public class MyClass
{
   public static void Main()
   {
      error e;
      // uncomment the next line to resolve this error
      // e.i = 0;
      System.Console.WriteLine( e.i );   // CS0170 because
                                         //e.i was never assigned
   }
}
```

Compiler Error CS0172

Article • 09/15/2021

Type of conditional expression cannot be determined because 'type1' and 'type2' implicitly convert to one another

In a conditional statement, you must be able to implicitly convert the types on either side of the `:` token. Also, there cannot be mutual implicit conversions; you only need one conversion.

The following sample generates CS0172:

C#

```
// CS0172.cs
public class Square
{
   public class Circle
   {
      public static implicit operator Circle(Square aa)
      {
         return null;
      }

      public static implicit operator Square(Circle aa)
      // using explicit resolves this error
      // public static explicit operator Square(Circle aa)
      {
         return null;
      }
   }

   public static void Main()
   {
      Circle aa = new Circle();
      Square ii = new Square();
      object o = (1 == 1) ? aa : ii;   // CS0172
      // the following cast would resolve this error
      // (1 == 1) ? aa : (Circle)ii;
   }
}
```

See also

- User-defined conversion operators

Compiler Error CS0173

08/08/2025

Type of conditional expression cannot be determined because there is no implicit conversion between 'type1' and 'type2'.

This error occurs when the compiler can't determine the type of a conditional expression because the two possible return values have no implicit conversion between them. This can happen with classes, value types, or any other types where there's no common type that both can be implicitly converted to.

Starting with C# 9, *target-typed conditional expressions* allow the compiler to use the target type (the type being assigned to) to resolve ambiguity in some cases. However, CS0173 still occurs when using `var` or when there's no target type to guide the compiler.

To resolve CS0173, you can:

- Provide an explicit target type (available in C# 9+):

  ```
  C#
  ```

  ```
  object result = condition ? value1 : value2;  // Works in C# 9.0+
  ```

- Use explicit casting:

  ```
  C#
  ```

  ```
  var result = condition ? (object)value1 : (object)value2;
  ```

- Ensure there's an implicit conversion between the types by adding conversion operators or using compatible types.

For more information, see User-defined conversion operators and Built-in numeric conversions.

Examples

Example 1: CS0173 with `var` (all C# versions)

The following example generates CS0173 because `var` provides no target type for the compiler to use:

```
C#
```

```
class Program
{
    static void Main()
    {
        // CS0173: Type of conditional expression can't be determined
        // because there is no implicit conversion between 'int' and 'string'.
        var result = true ? 100 : "ABC";
    }
}
```

To fix this, provide an explicit type:

C#

```
// Fix: Use explicit target type (C# 9.0+).
object result = true ? 100 : "ABC";  // OK in C# 9.0+

// Or use explicit casting (all versions).
var result = true ? (object)100 : (object)"ABC";
```

Example 2: Class conversion example

The following example shows CS0173 with custom classes:

C#

```
public class C {}

public class A
{
    // Uncomment to add implicit conversion from C to A.
    //public static implicit operator A(C c)
    //{
    //    return new A();
    //}
}

public class MyClass
{
    public static void F(bool b)
    {
        A a = new A();
        C c = new C();

        // CS0173: No implicit conversion between A and C.
        var result = b ? a : c;

        // Fix: Cast to common base type.
        object result2 = b ? (object)a : (object)c;
```

```
        // Or in C# 9.0+, provide target type.
        object result3 = b ? a : c;  // OK in C# 9.0+.
    }
}
```

Example 3: Version differences with nullable types

The behavior of conditional expressions has evolved across C# versions:

C#

```
class Program
{
    static void Main()
    {
        // This example shows how different C# versions handle the same code.

        // In C# 8.0 and earlier: CS8957 (feature not available).
        // In C# 9.0+: Compiles successfully.
        object? result = (1 == 0) ? null : null;
    }
}
```

> **Note**
>
> Starting with C# 9, target-typed conditional expressions allow the compiler to use the assignment target's type to resolve type ambiguity. In earlier versions, you might see error CS8957 instead of CS0173 when using language features not available in the current language version.

Compiler Error CS0174

Article • 09/15/2021

A base class is required for a 'base' reference

This error occurs only when the .NET runtime compiles the source code for the `System.Object` class, which is the only class that has no base class.

Compiler Error CS0175

Article • 09/15/2021

Use of keyword 'base' is not valid in this context

The base keyword must be used to specify a particular member of the base class. For more information, see Constructors.

The following sample generates CS0175:

C#

```
// CS0175.cs
using System;
class BaseClass
{
    public int TestInt = 0;
}

class MyClass : BaseClass
{
    public static void Main()
    {
        MyClass aClass = new MyClass();
        aClass.BaseTest();
    }

    public void BaseTest()
    {
        Console.WriteLine(base); // CS0175
        // Try the following line instead:
        // Console.WriteLine(base.TestInt);
        base = 9;   // CS0175

        // Try the following line instead:
        // base.TestInt = 9;
    }
}
```

Compiler Error CS0176

Article • 09/15/2021

Static member 'member' cannot be accessed with an instance reference; qualify it with a type name instead

Only a class name can be used to qualify a static variable; an instance name cannot be a qualifier. For more information, see Static Classes and Static Class Members.

The following sample generates CS0176:

C#

```
// CS0176.cs
public class MyClass2
{
    public static int num;
}

public class Test
{
    public static void Main()
    {
        MyClass2 mc2 = new MyClass2();
        int i = mc2.num;   // CS0176
        // try the following line instead
        // int i = MyClass2.num;
    }
}
```

Compiler Error CS0177

Article • 09/21/2022

The out parameter 'parameter' must be assigned to before control leaves the current method

A parameter marked with the out keyword was not assigned a value in the method body. For more information, see Passing Parameters

The following sample generates CS0177:

C#

```
// CS0177.cs
public class MyClass
{
   public static void Foo(out int i)   // CS0177
   {
   // uncomment the following line to resolve this error
   //   i = 0;
   }

   public static void Main()
   {
       int x = -1;
       Foo(out x);
   }
}
```

Compiler Error CS0179

Article • 09/15/2021

'member' cannot be extern and declare a body

When a class member is marked extern, it means that the member's definition is located in another file. Therefore, a class member marked as **extern** cannot be defined in the class. Either remove the `extern` keyword or delete the definition. For more information, see Methods.

The following sample generates CS0179:

C#

```
// CS0179.cs
public class MyClass
{
   public extern int ExternMethod(int aa)   // CS0179
   {
      return 0;
   }
   // try the following line instead
   // public extern int ExternMethod(int aa);

   public static void Main()
   {
   }
}
```

Compiler Error CS0180

Article • 09/15/2021

'member' cannot be both extern and abstract

The abstract and extern keywords are mutually exclusive. The `extern` keyword means that the member is defined outside the file, and **abstract** means that the implementation is provided in a derived class. For more information, see Methods.

The following sample generates CS0180:

C#

```
// CS0180.cs
namespace MyNamespace
{
   public class MyClass
   {
      public extern abstract int Foo(int a);   // CS0180

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0181

Article • 09/16/2022

Attribute constructor parameter has type, which is not a valid attribute parameter type

Example

The following sample generates CS0181:

C#

```
// CS0181.cs (12,6)

using System;
using System.Runtime.InteropServices;
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
unsafe class Attr : Attribute
{
    public Attr(delegate*<void> d) {}
}
unsafe class C
{
    [UnmanagedCallersOnly]
    [Attr(&M1)]
    static void M1()
    {
    }
}
```

The CLR currently does not allow generic attribute parameter types, so C# does not allow generic attribute parameter types.

Compiler Error CS0186

Article • 09/15/2021

Use of null is not valid in this context

The following sample generates CS0186:

C#

```
// CS0186.cs
using System;
using System.Collections;

class MyClass
{
   static void Main()
   {
      // Each of the following lines generates CS0186:
      foreach (int i in null) {}   // CS0186
      foreach (int i in (IEnumerable) null) { };   // CS0186
   }
}
```

Compiler Error CS0191

Article • 09/15/2021

Property or indexer 'name' cannot be assigned to -- it is read only

A readonly field can only take an assignment in a constructor or at declaration. For more information, see Constructors.

CS0191 also results if the `readonly` field is static and the constructor is not marked `static`.

Example

The following sample generates CS0191.

C#

```
// CS0191.cs
class MyClass
{
    public readonly int TestInt = 6;  // OK to assign to readonly field in
declaration

    MyClass()
    {
        TestInt = 11; // OK to assign to readonly field in constructor
    }

    public void TestReadOnly()
    {
        TestInt = 19;                  // CS0191
    }

    public static void Main()
    {
    }
}
```

Compiler Error CS0198

Article • 09/15/2021

Fields of static readonly field 'name' cannot be assigned to (except in a static constructor or a variable initializer)

A readonly variable must have the same static usage as the constructor in which you want to initialize it. For more information, see Static Constructors.

The following sample generates CS0198:

C#

```
// CS0198.cs
class MyClass
{
   public static readonly int TestInt = 6;

   MyClass()
   {
      TestInt = 11;   // CS0198, constructor is not static and readonly
field is
   }

   public static void Main()
   {
   }
}
```

Compiler error CS0200

Article • 10/26/2024

Property or indexer 'property' cannot be assigned to -- it is read only

An attempt was made to assign a value to a property, but the property does not have a set accessor or the assignment was outside of the constructor. Resolve the error by adding a set accessor. For more information, see How to declare and use read-write properties.

Examples

The following sample generates CS0200:

C#

```
// CS0200.cs
public class Example
{
    private int _mi;
    int I
    {
        get
        {
            return _mi;
        }
        // uncomment the set accessor and declaration for _mi
        /*
        set
        {
            _mi = value;
        }
        */
    }

    public static void Main()
    {
        Example example = new Example();
        example.I = 9;   // CS0200
    }
}
```

The following sample uses automatically implemented properties and object initializers and still generates CS0200:

C#

```
// CS0200.cs
public class Example
{
    int I
    {
        get;
        // uncomment the set accessor and declaration
        //set;
    }

    public static void Main()
    {
        var example = new Example
        {
            I = 9   // CS0200
        };
    }
}
```

To assign to a property or indexer 'property' that's read-only, add a set accessor or assign the value in the object's constructor.

C#

```
public class Example
{
    int I { get; }

    public Example()
    {
        I = -7;
    }
}
```

Compiler Error CS0201

Article • 03/31/2022

Only assignment, call, increment, decrement, and new object expressions can be used as a statement

The compiler generates an error when it encounters an invalid statement. An invalid statement is any line or series of lines ending in a semicolon that does not represent an assignment (=), method call (), new, -- or ++ operation. For more information, see Statements and Operators and expressions.

Example 1

The following sample generates CS0201 because 2 * 3 is an expression, not a statement. To make the code compile, try assigning the value of the expression to a variable.

C#

```
// CS0201.cs
public class MainClass
{
   public static void Main()
   {
      2 * 3;   // CS0201
      // Try the following line instead.
      //   int i = 2 * 3;
   }
}
```

Example 2

The following sample generates CS0201 because checked by itself is not a statement, even though it is parameterized by an increment operation.

C#

```
// CS0201_b.cs
// compile with: /target:library
public class MyList<T>
{
   public void Add(T x)
   {
      int i = 0;
      if ( (object)x == null)
```

```
        {
            checked(i++);    // CS0201

            // OK
            checked {
                i++;
            }
        }
    }
}
```

See also

- C# Compiler Errors

Compiler Error CS0202

Article • 09/15/2021

foreach requires that the return type 'type' of 'type.GetEnumerator()' must have a suitable public MoveNext method and public Current property

A GetEnumerator function, used to enable the use of foreach statements, cannot return a pointer or array; it must return an instance of a class that is able to act as an enumerator. The proper requirements to serve as an enumerator include a public Current property and a public MoveNext method.

> **Note**
>
> In C# 2.0, the compiler will automatically generate Current and MoveNext for you. For more information, see the code example in Generic Interfaces.

The following sample generates CS0202:

C#

```
// CS0202.cs

public class C1
{
   public int Current
   {
      get
      {
         return 0;
      }
   }

   public bool MoveNext ()
   {
      return false;
   }

   public static implicit operator C1 (int c1)
   {
      return 0;
   }
}

public class C2
{
   public int Current
   {
```

```
      get
      {
         return 0;
      }
   }

   public bool MoveNext ()
   {
      return false;
   }

   public C1[] GetEnumerator ()
   // try the following line instead
   // public C1 GetEnumerator ()
   {
      return null;
   }
}

public class MainClass
{
   public static void Main ()
   {
      C2 c2 = new C2();

      foreach (C1 x in c2)   // CS0202
      {
         System.Console.WriteLine(x.Current);
      }
   }
}
```

Compiler Error CS0204

07/22/2025

Only 65534 locals are allowed

The limit of 65534 local variables has been exceeded in a method.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

Compiler Error CS0205

Article • 09/15/2021

Cannot call an abstract base member: 'method'

You cannot call an abstract method because it does not have a method body. For more information, see Abstract and Sealed Classes and Class Members.

The following sample generates CS0205:

C#

```
// CS0205.cs
abstract public class MyClass
{
   abstract public void M();
}

public class MyClass2 : MyClass
{
   public override void M()
   {
      base.M();   // CS0205, delete this line
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0220

Article • 08/24/2022

The operation overflows at compile time in checked mode

An operation was detected by checked, which is the default for constant expressions, and a data loss resulted. Either correct the inputs to the assignment or use unchecked to resolve this error. For more information, see the checked and unchecked statements article.

The following sample generates CS0220:

C#

```
// CS0220.cs
using System;

class TestClass
{
   const int x = 1000000;
   const int y = 1000000;

   public int MethodCh()
   {
      int z = (x * y);   // CS0220
      return z;
   }

   public int MethodUnCh()
   {
      unchecked
      {
         int z = (x * y);
         return z;
      }
   }

   public static void Main()
   {
      TestClass myObject = new TestClass();
      Console.WriteLine("Checked  : {0}", myObject.MethodCh());
      Console.WriteLine("Unchecked: {0}", myObject.MethodUnCh());
   }
}
```

Compiler Error CS0221

Article • 08/24/2022

Constant value 'value' cannot be converted to a 'type' (use 'unchecked' syntax to override)

An assignment operation that would result in a data loss was detected by checked, which is on by default for constant expressions. Either correct the assignment or use unchecked to resolve this error. For more information, see the checked and unchecked statements article.

The following sample generates CS0221:

C#

```
// CS0221.cs
public class MyClass
{
   public static void Main()
   {
      // unchecked
      // {
         int a = (int)0xFFFFFFFF;   // CS0221
         a++;
      // }
   }
}
```

Compiler Error CS0226

Article • 09/15/2021

An __arglist expression may only appear inside of a call or new expression.

The unsupported keyword `__arglist` can only appear in a method call or a new expression.

Example

The following code generates CS0226:

C#

```
// cs0226.cs
using System;

public class C
    {
    public static int Main ()
        {
        __arglist(1,"This is a string"); // CS0226
        return 0;
        }
    }
```

Compiler Error CS0228

Article • 09/15/2021

'type' does not contain a definition for 'member', or it is not accessible

You may have replaced the system version of **System.Hashtable**, **System.String**, or **System.Array**, and the replacement does not contain `member`.

Otherwise, you may need to repair or reinstall Visual Studio.

Compiler Error CS0229

Article • 09/15/2021

Ambiguity between 'member1' and 'member2'

Members of different interfaces have the same name. If you want to keep the same names, you must qualify the names. For more information, see Interfaces.

> **Note**
>
> In some cases, this ambiguity can be resolved by providing an explicit prefix to the identifier via a using alias.

Example

The following example generates CS0229:

C#

```
// CS0229.cs

interface IList
{
    int Count
    {
        get;
        set;
    }

    void Counter();
}

interface ICounter
{
    double Count
    {
        get;
        set;
    }
}

interface IListCounter : IList, ICounter {}

class MyClass
{
    void Test(IListCounter x)
    {
```

```
        x.Count = 1;  // CS0229
        // Try one of the following lines instead:
        // ((IList)x).Count = 1;
        // or
        // ((ICounter)x).Count = 1;
    }

    public static void Main() {}
}
```

Compiler Error CS0230

Article • 10/19/2023

Type and identifier are both required in a foreach statement

A foreach statement was poorly formed.

The following sample generates CS0230:

C#

```
// CS0230.cs
class MyClass
{
   public static void Main()
   {
      int[] myarray = new int[3] {1,2,3};

      foreach (int in myarray)   // CS0230
      {
         Console.WriteLine(x);
      }
   }
}
```

and the sample below presents the same code, but with no CS0230 error:

C#

```
class MyClass
{
   public static void Main()
   {
      int[] myarray = new int[3] {1,2,3};

      foreach (int x in myarray)  // Both type (int) and indentifier (x) are
specified
      {
         Console.WriteLine(x);
      }
   }
}
```

Compiler Error CS0236

07/16/2025

A field initializer cannot reference the non-static field, method, or property 'name'.

Instance fields cannot be used to initialize other instance fields outside a method.

Why this error occurs

The compiler enforces this restriction because of how object initialization works in C#. When an object is created, field initializers are processed before any constructor code runs. During this phase:

1. **Field dependency restriction**: While field initializers are processed in lexical order within a single file, the rule prevents dependencies between fields to allow class authors to rearrange fields without introducing compiler errors. For partial classes, the order of field initializers across different source files isn't specified.

2. **Object is not fully constructed**: When field initializers run, the object instance is in an incomplete state. Allowing references between fields during this phase could lead to accessing uninitialized memory or unpredictable behavior.

3. **Compiler safety**: This restriction prevents potential runtime errors and ensures predictable object construction behavior.

The compiler detects this pattern during compilation and reports CS0236 to prevent these potential issues.

To correct this error

1. **Move initialization to constructor**: Perform the initialization inside an instance constructor where all fields are guaranteed to be available.

2. **Use static fields**: If the referenced field doesn't need to be instance-specific, consider making it static.

3. **Use default values**: Initialize fields with literal values or expressions that don't reference other instance members.

4. **Lazy initialization**: Use properties with backing fields for complex initialization logic that depends on other instance members.

Example

The following sample generates CS0236, and shows how to fix it:

C#

```
public class MyClass
{
    public int i = 5;

    // CS0236: Field initializer cannot reference instance field 'i'
    // This restriction allows class authors to rearrange fields without compiler
errors
    public int j = i;  // CS0236

    public MyClass()
    {
        // This works because both fields are guaranteed to be initialized
        // before constructor code runs
        j = i;
    }
}
```

Additional examples

The following examples demonstrate different scenarios where CS0236 occurs:

C#

```
public class Examples
{
    private string name = "Default";

    // CS0236: Cannot reference instance field in initializer
    private string displayName = name.ToUpper();

    // CS0236: Cannot reference instance method in initializer
    private int length = GetNameLength();

    // CS0236: Cannot reference instance property in initializer
    private string formatted = FormattedName;

    public string FormattedName => $"Name: {name}";

    private int GetNameLength() => name.Length;

    public Examples()
    {
        // All of these work in the constructor:
        displayName = name.ToUpper();
```

```
        length = GetNameLength();
        formatted = FormattedName;
    }
}
```

Compiler Error CS0238

Article • 09/15/2021

'member' cannot be sealed because it is not an override

sealed was used on a member that was not also marked with override. For more information, see Inheritance.

The following sample generates CS0238:

C#

```
// CS0238.cs
abstract class MyClass
{
   public abstract void f();
}

class MyClass2 : MyClass
{
   public static void Main()
   {
   }

   public sealed void f() // CS0238
   // Try the following definition instead:
   // public override sealed void f()
   {
   }
}
```

Compiler Error CS0239

Article • 08/18/2022

'member' : cannot override inherited member 'inherited member' because it is sealed

A member cannot override a sealed inherited member. For more information, see Inheritance.

The following sample generates CS0239:

C#

```
// CS0239.cs
abstract class MyClass
{
   public abstract void f();
}

class MyClass2 : MyClass
{
   public static void Main()
   {
   }

   public override sealed void f()
   {
   }
}

class MyClass3 : MyClass2
{
   public override void f()   // CS0239
   // Try the following definition instead:
   // public new void f()
   {
   }
}
```

Compiler Error CS0241

Article • 09/21/2022

Default parameter specifiers are not permitted

Method parameters cannot have default values. Use method overloads if you want to achieve the same effect.

Example

The following sample generates CS0241. In addition, the sample shows how to simulate, with overloading, a method with default arguments.

C#

```
// CS0241.cs
public class A
{
   public void Test(int i = 9) {}   // CS0241
}

public class B
{
   public void Test() { Test(9); }
   public void Test(int i)  {}
}

public class C
{
   public static void Main()
   {
      B x = new B();
      x.Test();
   }
}
```

Compiler Error CS0247

Article • 09/15/2021

Cannot use a negative size with stackalloc

A negative number was passed to a stackalloc statement.

The following sample generates CS0247:

C#

```
// CS0247.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int *p = stackalloc int[-30];   // CS0247
   }
}
```

Compiler Error CS0249

Article • 09/15/2021

Do not override object.Finalize. Instead, provide a destructor.

Use finalizer syntax to specify instructions to execute when your object is destroyed.

The following sample generates CS0249:

C#

```
// CS0249.cs
class MyClass
{
   protected override void Finalize()   // CS0249
   // try the following line instead
   // ~MyClass()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0250

Article • 11/01/2023

Do not directly call your base class Finalize method. It is called automatically from your destructor.

A program cannot attempt to force cleanup of base class resources.

See Finalizers for more information. This message uses the previous term for finalizer: *destructor*.

The following sample generates CS0250

C#

```
// CS0250.cs
class B
{
}

class C : B
{
   ~C()
   {
      base.Finalize();   // CS0250
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0255

Article • 04/22/2023

stackalloc may not be used in a catch or finally block

You cannot use the stackalloc operator in a catch or finally block. For more information, see Exceptions and Exception Handling.

The following sample generates CS0255:

C#

```
// CS0255.cs
// compile with: /unsafe
using System;

public class TestTryFinally
{
   public static unsafe void Test()
   {
      int i = 123;
      string s = "Some string";
      object o = s;

      try
      {
         // Conversion is not valid; o contains a string not an int
         i = (int) o;
      }
      finally
      {
         Console.Write("i = {0}", i);
         int* fib = stackalloc int[100];   // CS0255
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0266

Article • 09/15/2021

Cannot implicitly convert type 'type1' to 'type2'. An explicit conversion exists (are you missing a cast?)

This error occurs when your code tries to convert between two types that cannot be implicitly converted, but where an explicit conversion is available. For more information, see Casting and Type Conversions.

The following code shows examples that generate CS0266:

C#

```
// CS0266.cs
class MyClass
{
    public static void Main()
    {
        // You cannot implicitly convert a double to an integer.
        double d = 3.2;

        // The following line causes compiler error CS0266.
        int i1 = d;

        // However, you can resolve the error by using an explicit
conversion.
        int i2 = (int)d;

        // You cannot implicitly convert an object to a class type.
        object obj = new MyClass();

        // The following assignment statement causes error CS0266.
        MyClass myClass = obj;

        // You can resolve the error by using an explicit conversion.
        MyClass c = (MyClass)obj;

        // You cannot implicitly convert a base class object to a derived
class type.
        MyClass mc = new MyClass();
        DerivedClass dc = new DerivedClass();

        // The following line causes compiler error CS0266.
        dc = mc;

        // You can resolve the error by using an explicit conversion.
        dc = (DerivedClass)mc;
    }
}
```

```
class DerivedClass : MyClass
{
}
```

Compiler Error CS0268

Article • 09/15/2021

Imported type 'type' is invalid. It contains a circular base class dependency.

This error occurs if a type imported from another language has a circular base class dependency. Such a type cannot be used in a C# program. To resolve this error, check types imported from other languages in any referenced assemblies or modules.

Compiler Error CS0269

Article • 09/21/2022

Use of unassigned out parameter 'parameter'

The compiler could not verify that the out parameter was assigned a value before it was used; its value may be undefined when assigned. Be sure to assign a value to `out` parameters in the called method before accessing the value. If you need to use the value of the variable passed in, use a `ref` parameter instead. For more information, see Method Parameters.

Example 1

The following sample generates CS0269:

C#

```
// CS0269.cs
class C
{
    public static void F(out int i)
    // One way to resolve the error is to use a ref parameter instead
    // of an out parameter.
    //public static void F(ref int i)
    {
        // The following line causes a compiler error because no value
        // has been assigned to i.
        int k = i;  // CS0269
        i = 1;
        // The error does not occur if the order of the two previous
        // lines is reversed.
    }

    public static void Main()
    {
        int myInt = 1;
        F(out myInt);
        // If the declaration of method F is changed to require a ref
        // parameter, ref must be specified in the call as well.
        //F(ref myInt);
    }
}
```

Example 2

This could also occur if the initialization of a variable occurs in a try block, which the compiler is unable to verify will execute successfully:

C#

```
// CS0269b.cs
class C
{
    public static void F(out int i)
    {
        try
        {
            // Assignment occurs, but compiler can't verify it
            i = 1;
        }
        catch
        {
        }

        int k = i;  // CS0269
        i = 1;
    }

    public static void Main()
    {
        int myInt;
        F(out myInt);
    }
}
```

Compiler Error CS0271

Article • 09/15/2021

The property or indexer 'property/indexer' cannot be used in this context because the get accessor is inaccessible

This error occurs when you try to access an inaccessible `get` accessor. To resolve this error, increase the accessibility of the accessor, or change the calling location. For more information, see Accessor Accessibility and Properties.

The following example generates CS0271:

C#

```
// CS0271.cs
public class MyClass
{
   public int Property
   {
      private get { return 0; }
      set { }
   }

   public int Property2
   {
      get { return 0; }
      set { }
   }
}

public class Test
{
   public static void Main(string[] args)
   {
      MyClass c = new MyClass();
      int a = c.Property;   // CS0271
      int b = c.Property2;   // OK
   }
}
```

Compiler Error CS0272

Article • 09/21/2022

The property or indexer 'property/indexer' cannot be used in this context because the set accessor is inaccessible.

This error occurs when the `set` accessor is not accessible to the program code.

To correct this error

Increase the accessibility of the accessor, or change the calling location. For more information, see Restricting Accessor Accessibility.

Example

The following example generates CS0272:

C#

```
// CS0272.cs
public class MyClass
{
    public int Property
    {
        get { return 0; }
        private set { }
    }
}

public class Test
{
    static void Main()
    {
        MyClass c = new MyClass();
        c.Property = 10;      // CS0272
        // To resolve, remove the previous line
        // or use an appropriate modifier on the set accessor.
    }
}
```

Compiler error CS0273

Article • 09/22/2022

The accessibility modifier of the 'property_accessor' accessor must be more restrictive than the property or indexer 'property'

The accessibility modifier of the set/get accessor must be more restrictive than the property or indexer 'property/indexer'

This error occurs when the accessibility of the accessor you declared isn't less restrictive than the accessibility of the property or indexer.

To correct this error

Use the appropriate access modifier on either the property or the accessor. For more information, see Restricting Accessor Accessibility and Accessors.

Example

This sample contains an internal property with an internal set method. The following sample generates CS0273.

C#

```
// CS0273.cs
// compile with: /target:library
public class MyClass
{
   internal int Property
   {
      get { return 0; }
      internal set {}   // CS0273
      // try the following line instead
      // private set {}
   }
}
```

Compiler Error CS0274

Article • 09/15/2021

Cannot specify accessibility modifiers for both accessors of the property or indexer 'property/indexer'

This error occurs when you declare a property or indexer with access modifiers on both its accessors. To resolve this error, use an access modifier on only one of the two accessors. For more information, see Accessor Accessibility.

The following example generates CS0274:

C#

```
// CS0274.cs
public class MyClass
{
    public int Property   // CS0274
    {
        public get { return 0; }
        protected set { }
    }
}
```

Compiler Error CS0275

Article • 09/15/2021

'accessor': accessibility modifiers may not be used on accessors in an interface

This error occurs when you use an access modifier on any one of the accessors of a property or indexer in an interface. To resolve, remove the access modifier.

Example

The following example generates CS0275:

C#

```
// CS0275.cs
public interface MyInterface
{
    int Property
    {
        get;
        internal set;   // CS0275
    }
}
```

Compiler Error CS0276

Article • 09/15/2021

'property/indexer': accessibility modifiers on accessors may only be used if the property or indexer has both a get and a set accessor

This error occurs when you declare a property or indexer with one accessor only, and use an access modifier on the accessor. To resolve, remove the access modifier or add another accessor.

Example

The following example generates CS0276:

C#

```
// CS0276.cs
public class MyClass
{
    public int Property
    {
        protected set { }   // CS0276
    }
    public int Property2
    {
        internal get { }   // CS0276
    }
}
```

Compiler Error CS0281

Article • 09/15/2021

Friend access was granted to 'AssemblyName1', but the output assembly is named 'AssemblyName2'. Try adding a reference to 'AssemblyName1' or changing the output assembly name to match.

Friend access is a new common language runtime (CLR) feature that enables an assembly to see another assembly's non-public types. This error occurs when the assembly granting friend access specifies the wrong name for the grantee assembly. For more information, see Friend Assemblies.

Example 1

The following sequence of code samples will generate CS0281.

The files used to create the strong named assemblies are generated as follows:

- sn -d CS0281.snk
- sn -k CS0281.snk
- sn -i CS0281.snk CS0281.snk
- sn -pc CS0281.snk key.publickey
- sn -tp key.publickey

C#

```
// CS0281.cs
// compile with: /target:library /keyfile:CS0281.snk
public class A {}
```

Example 2

C#

```
// CS0281_b.cs
// compile with: /target:library /keyfile:CS0281.snk /reference:CS0281.dll
[assembly:System.Runtime.CompilerServices.InternalsVisibleTo("CS0281,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100 4b
2d4d56af7c50be2fcbbf97cb880b9e73ad84467a587191fef63aadc118a96cecf9d508cd679c
907b6e20f71684300bdc2c0a851019af0c96b29bf8f1339753276041aefd67db46139e6348b3
```

```
a12f29537b4dc6c2c19829df2c9ed6803f3c63c3b84cfa2728849386aea575c543a5f70fa857
93d2946f15f7fe1ccb0c5e8fe0")]
class B : A {}
```

Example 3

The following sample generates CS0281.

Notice that this sample creates an output file with the same name as the output file in the first sample. To resolve, do not change the assembly attributes of the component and add class C.

C#

```
// CS0281_c.cs
// compile with: /target:library /out:CS0281.dll /keyfile:CS0281.snk
/reference:CS0281_b.dll
// CS0281 expected
[assembly:System.Reflection.AssemblyVersion("3")]
[assembly:System.Reflection.AssemblyCulture("en-us")]
class C : B {}
public class A {}
```

Compiler Error CS0283

Article • 09/15/2021

The type 'type' cannot be declared const

The type specified in a constant declaration must be `byte`, `sbyte`, `ushort`, `short`, `uint`, `int`, `ulong`, `long`, `char`, `float`, `double`, `decimal`, `bool`, `string`, an enum type, or a reference type that is assigned a value of `null`. Each constant expression must yield a value of the target type or of a type that is implicitly convertible to the target type.

Example

The following example generates CS0283.

C#

```
// CS0283.cs
struct MyTest
{
}
class MyClass
{
    // To resolve the error but retain the "const-ness",
    // change const to readonly.
    const MyTest test = new MyTest();   // CS0283

    public static int Main() {
        return 1;
    }
}
```

Compiler Error CS0316

Article • 09/15/2021

The parameter name 'name' conflicts with an automatically-generated parameter name.

Reserved words cannot be used as parameter names. In the example that follows, `value` is a reserved word in the context of a default property or indexer accessor.

To correct this error

1. Change the name of the parameter.

Example

The following code generates CS0316:

C#

```
// cs0316.cs
// Compile with: /target:library
public class Test
{
    public int this[int value] // CS0316
    {
        get { return 1; }
        set { }
    }
}
```

See also

- Indexers
- C# Keywords

Compiler Error CS0401

Article • 08/01/2024

The new() constraint must be the last restrictive constraint specified

When using multiple constraints, list all other constraints before the new() constraint.

Example

The following sample generates CS0401.

C#

```
// CS0401.cs
// compile with: /target:library
using System;
 class C<T> where T : new(), IDisposable {}  // CS0401

class D<T> where T : IDisposable
{
   static void F<U>() where U : new(), IDisposable{}   // CS0401
}
```

Compiler Error CS0405

Article • 09/15/2021

Duplicate constraint 'constraint' for type parameter 'type parameter'

Two of the constraints on the generic declaration are identical. To get rid of the error, remove the duplicate.

The following sample generates CS0405:

C#

```
// CS0405.cs
interface I
{
}

class C<T> where T : I, I  // CS0405.cs
{
}
```

Compiler Error CS0406

Article • 09/15/2021

The class type constraint 'constraint' must come before any other constraints

When a generic type or method has a class type constraint, that constraint must be listed first. To avoid this error, move the class type constraint to the beginning of the constraint list.

Example

The following sample generates CS0406.

C#

```
// CS0406.cs
// compile with: /target:library
interface I {}
class C {}
class D<T> where T : I, C {}   // CS0406
class D2<T> where T : C, I {}   // OK
```

Compiler Error CS0407

Article • 09/15/2021

'return-type method' has the wrong return type

The method was not compatible with the delegate type. The argument types matched, but the return type was not the correct return type for that delegate. To avoid this error, use a different method, change the method's return type, or change the delegate's return type.

Example

The following sample generates CS0407:

C#

```
// CS0407.cs
public delegate int MyDelegate();

class C
{
    MyDelegate d;

    public C()
    {
        d = new MyDelegate(F);  // OK: F returns int
        d = new MyDelegate(G);  // CS0407 - G doesn't return int
    }

    public int F()
    {
        return 1;
    }

    public void G()
    {
    }

    public static void Main()
    {
        C c1 = new C();
    }
}
```

Compiler Error CS0409

Article • 09/15/2021

A constraint clause has already been specified for type parameter 'type parameter'. All of the constraints for a type parameter must be specified in a single where clause.

Multiple constraint clauses (where clauses) were found for a single type parameter. Remove the extraneous where clause, or correct the where clauses so that a unique type parameter in each clause.

C#

```
// CS0409.cs
interface I
{
}

class C<T1, T2> where T1 : I where T1 : I  // CS0409 - T1 used twice
{
}
```

Compiler Error CS0410

Article • 09/15/2021

No overload for 'method' has the correct parameter and return types

This error occurs if you try to instantiate a delegate with a function that has the wrong parameter types. The parameter types of the delegate must match the function that you are assigning to the delegate.

Example

The following example generates CS0410:

C#

```
// CS0410.cs
// compile with: /langversion:ISO-1

class Test
{
    delegate void D(double d );
    static void F(int i) { }

    static void Main()
    {
        D d = new D(F);  // CS0410
    }
}
```

> **Note**
>
> This compiler error is no longer used in Roslyn. The previous example generates CS0123 when compiled with Roslyn.

Compiler Error CS0411

Article • 09/15/2021

The type arguments for method 'method' cannot be inferred from the usage. Try specifying the type arguments explicitly.

This error occurs if you call a generic method without explicitly providing the type arguments and the compiler cannot infer which type arguments are intended. To avoid this error, add the intended type arguments in angle brackets.

Example 1

The following sample generates CS0411:

C#

```
// CS0411.cs
class C
{
    void G<T>()
    {
    }

    public static void Main()
    {
        G();  // CS0411
        // Try this instead:
        // G<int>();
    }
}
```

Example 2

Other possible error cases include when the parameter is `null`, which has no type information:

C#

```
// CS0411b.cs
class C
{
    public void F<T>(T t) where T : C
    {
    }
```

```
    public static void Main()
    {
        C c = new C();
        c.F(null);  // CS0411
    }
}
```

Compiler Error CS0418

Article • 09/15/2021

'class name': an abstract class cannot be sealed or static

An abstract class cannot be used to create objects unless it is derived from, so it makes no sense to be sealed. An abstract class cannot meaningfully be static either; abstract classes are designed to support an object hierarchy that will use the abstract class as a base.

Example

The following sample generates CS0418:

C#

```
// CS0418.cs
public abstract sealed class C  // CS0418
{
}

sealed static class S  // CS0418
{
}

public class MyClass
{
    public static void Main()
    {
    }
}
```

Compiler Error CS0423

Article • 09/15/2021

Since 'class' has the ComImport attribute, 'method' must be extern or abstract

Specifying the ComImport attribute implies that the implementation for the class is to be imported from a COM module. Additional methods may not be defined.

The following sample generates CS0423:

C#

```
// CS0423.cs

using System.Runtime.InteropServices;

[
  ComImport,
  Guid("7ab770c7-0e23-4d7a-8aa2-19bfad479829")
]
class ImageProperties
{
  public static void Main()  // CS0423
  {
    ImageProperties i = new ImageProperties();
  }
}
```

Compiler Error CS0424

Article • 09/15/2021

'class': a class with the ComImport attribute cannot specify a base class

Specifying the ComImportAttribute attribute implies that the implementation for the class is to be imported from a COM module. Additional methods or fields inherited from the base class are not allowed to be added to the implementation defined in the COM module.

The following sample generates CS0424:

C#

```
// CS0424.cs
// compile with: /target:library
using System.Runtime.InteropServices;
public class A {}

[ ComImport, Guid("7ab770c7-0e23-4d7a-8aa2-19bfad479829") ]
class B : A {}   // CS0424 error
```

Compiler Error CS0426

Article • 10/27/2021

The type name 'identifier' does not exist in the type 'type'

This error occurs when you reference a type nested within another type, but no such nested type exists. This can occur if you mistype the name of the nested type. Check the spelling of the names used, and verify that the enclosing type has the expected member.

The following sample generates CS0426 because class C has no nested type A:

C#

```
// CS0426.cs

class C
{
    // No nested types are declared.
}

class D
{
   public static void Main()
   {
       C c = new C();
       // Attempt to reference a nested type A:
       C.A a; // CS0426 because no such type C.A
   }
}
```

See also

- The C# type system

Compiler Error CS0428

Article • 09/15/2021

Cannot convert method group 'Identifier' to non-delegate type 'type'. Did you intend to invoke the method?

This error occurs when converting a method group to a non-delegate type, or attempting to invoke a method without using parentheses.

Example

The following sample generates CS0428:

C#

```
// CS0428.cs
namespace ConsoleApplication1
{
    class Program
    {
        delegate int Del1();
        delegate object Del2();

        static void Main(string[] args)
        {
            ExampleClass ec = new ExampleClass();

            // The following assignment statement causes compiler error
CS0428.
            // It attempts to assign the address of Method1 to an integer
variable.
            // You can only assign the address to an appropriate delegate
type.
            int i = ec.Method1;

            // Del1 is a delegate type that is appropriate for a method like
            // Method1 that returns an int. The following assignment
statement
            // does not cause an error.
            Del1 d1 = ec.Method1;

            // You can invoke Method1 and assign the int that is returned to
            // integer variable i.
            i = ec.Method1();

            // The following assignment statement causes compiler error
CS0428.
            // It attempts to assign the address of Method2 to an instance
of
```

```
            // ExampleClass. You can only assign the address to a delegate
type.
            ec = ExampleClass.Method2;

            // Del2 is a delegate type that is appropriate for a method like
            // Method2 that returns an instance of a class. The following
assignment
            // statement does not cause an error.
            Del2 d2 = ExampleClass.Method2;

            // Similarly, you can invoke Method2 and assign the result
returned to ec.
            ec = ExampleClass.Method2();
        }
    }

    public class ExampleClass
    {
        public int Method1() { return 1; }
        public static ExampleClass Method2() { return null; }
    }
}
```

Compiler Error CS0430

08/05/2025

The extern alias 'alias' was not specified in a /reference option

This error occurs when extern Alias is encountered but Alias was not specified as a reference on the command line. To resolve CS0430, compile with **/reference**.

Example 1

C#

```
// CS0430_a.cs
// compile with: /target:library
public class MyClass
{
    public static void M() { /*...*/ }
}
```

Example 2

Compiling with **/reference:MyType=cs0430_a.dll** to refer to the DLL created in the previous sample resolves this error. The following sample generates CS0430.

C#

```
// CS0430_b.cs
extern alias MyType;   // CS0430
public class Test
{
   public static void Main() { MyType::MyClass.M(); }
}
```

See also

- extern alias

Compiler Error CS0433

07/18/2025

The type TypeName1 exists in both TypeName2 and TypeName3

Two different assemblies referenced in your application contain the same namespace and type, which produces ambiguity.

To resolve this error, use the `extern alias` feature with project reference aliases or do not reference one of your assemblies. You can also use the alias feature of the (**References**) compiler option when compiling directly with the C# compiler.

This error can also occur if:

- The `@ Page` directive has a `CodeFile` attribute when it should be a `CodeBehind` attribute.
- Code is placed in an *App_Code* folder that shouldn't reside there.

How to identify the conflicting assemblies

The full error message shows which assemblies contain the conflicting type. The message format is:

```
error CS0433: The type 'N.C' exists in both 'A, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null' and 'B, Version=0.0.0.0, Culture=neutral,
PublicKeyToken=null'
```

In this example, the conflicting assemblies are **A** and **B**. The assembly names appear in single quotes before the version information. Use these assembly names to determine which references are causing the conflict and decide how to resolve it.

Examples

This code creates the DLL with the first copy of the ambiguous type.

C#

```
// CS0433_1.cs in CS0433_1.csproj
// compile with: dotnet build or /target:library
namespace TypeBindConflicts
{
```

```
    public class AggPubImpAggPubImp { }
}
```

This code creates the DLL with the second copy of the ambiguous type.

C#

```
// CS0433_2.cs in CS0433_2.csproj
// compile with: dotnet build or /target:library
namespace TypeBindConflicts
{
    public class AggPubImpAggPubImp { }
}
```

So, when consuming these two libraries (`CS0433_1.dll` and `CS0433_2.dll`) in the project, using the `AggPubImpAggPubImp` type will be ambiguous and will lead to compiler error `CS0433`.

XML

```
<!-- CS0433_3.csproj -->
<ProjectReference Include="..\CS0433_1\CS0433_1.csproj" />
<ProjectReference Include="..\CS0433_2\CS0433_2.csproj" />
```

C#

```
// CS0433_3.cs in CS0433_3.csproj
// compile with: dotnet build or /reference:cs0433_1.dll /reference:cs0433_2.dll
using TypeBindConflicts;

public class Test
{
    public static void Main()
    {
        AggPubImpAggPubImp n6 = new AggPubImpAggPubImp();   // CS0433
    }
}
```

The following example shows how you can use the `extern alias` feature with project references to resolve this CS0433 error. This is the recommended approach for modern .NET projects.

Step 1: Add an alias to one of the project references

First, modify your project file to add an alias to one of the conflicting project references:

XML

```
<!-- CS0433_4.csproj -->
<ProjectReference Include="..\CS0433_1\CS0433_1.csproj">
  <Aliases>CustomTypes</Aliases>
</ProjectReference>
<ProjectReference Include="..\CS0433_2\CS0433_2.csproj" />
```

Step 2: Use the extern alias in your code

Then, use the `extern alias` directive and qualified type names to distinguish between the two types:

C#

```
// CS0433_4.cs in CS0433_4.csproj
// compile with: dotnet build or /reference:cs0433_1.dll
/reference:CustomTypes=cs0433_2.dll
extern alias CustomTypes;
using TypeBindConflicts;

public class Test
{
   public static void Main()
   {
      // AggPubImpAggPubImp taken from CS0433_2.dll (no alias, default global
namespace)
      AggPubImpAggPubImp n6 = new AggPubImpAggPubImp();

      // AggPubImpAggPubImp taken from CS0433_1.dll (via CustomTypes alias)
      CustomTypes.TypeBindConflicts.AggPubImpAggPubImp n7 =
          new CustomTypes.TypeBindConflicts.AggPubImpAggPubImp();
   }
}
```

Compiler Error CS0434

Article • 09/15/2021

The namespace NamespaceName1 in NamespaceName2 conflicts with the type TypeName1 in NamespaceName3

This error occurs when an imported type and an imported nested namespace have the same fully qualified name. When that name is referenced, the compiler is unable to distinguish between the two. If you can change the imported source code, you can resolve the error by changing the name of either the type or the namespace so that both are unique within the assembly.

The following code generates error CS0434.

Example 1

This code creates the first copy of the type with the identical fully qualified name.

C#

```
// CS0434_1.cs
// compile with: /t:library
namespace TypeBindConflicts
{
    namespace NsImpAggPubImp
    {
        public class X { }
    }
}
```

Example 2

This code creates the second copy of the type with the identical fully qualified name.

C#

```
// CS0434_2.cs
// compile with: /t:library
namespace TypeBindConflicts {
    // Conflicts with another import (import2.cs).
    public class NsImpAggPubImp { }
    // Try this instead:
    // public class UniqueClassName { }
}
```

Example 3

This code references the type with the identical fully qualified name.

C#

```
// CS0434.cs
// compile with: /r:cs0434_1.dll /r:cs0434_2.dll
using TypeBindConflicts;
public class Test
{
    public TypeBindConflicts.NsImpAggPubImp.X n2 = null; // CS0434
}
```

Compiler Error CS0438

Article • 09/15/2021

The type 'type' in 'module_1' conflicts with the namespace 'namespace' in 'module_2'.

This error occurs when a type in a source file conflicts with a namespace in another source file. This typically happens when one or both come from an added module. To resolve, rename the type or the namespace that caused the conflict.

The following example generates CS0438:

Compile this file first:

C#

```
// CS0438_1.cs
// compile with: /target:module
public class Util
{
   public class A { }
}
```

Then compile this file:

C#

```
// CS0438_2.cs
// compile with: /target:module
namespace Util
{
   public class A { }
}
```

And then compile this file:

C#

```
// CS0438_3.cs
// compile with: /addmodule:CS0438_1.netmodule /addmodule:CS0438_2.netmodule
using System;
public class Test
{
   public static void Main() {
      Console.WriteLine(typeof(Util.A));   // CS0438
   }
}
```

Compiler Error CS0439

Article • 09/15/2021

An extern alias declaration must precede all other elements defined in the namespace

This error occurs when an `extern` declaration comes after something else, such as a `using` declaration, in the same namespace. The `extern` declarations must come before all other namespace elements.

Example

The following example generates CS0439:

C#

```
// CS0439.cs
using System;

extern alias MyType;   // CS0439
// To resolve the error, make the extern alias the first line in the file.

public class Test
{
    public static void Main()
    {
    }
}
```

Compiler Error CS0441

Article • 09/15/2021

'class': a class cannot be both static and sealed

This error occurs when you declare a class that is both static and sealed. Static classes are inherently sealed, so the sealed modifier is not necessary. To resolve, use one modifier only.

The following example generates CS0441:

C#

```
// CS0441.cs
sealed static class MyClass { }   // CS0441
```

Compiler Error CS0442

Article • 09/15/2021

'Property': abstract properties cannot have private accessors

This error occurs when you use the access modifier "private" to modify an abstract accessor. To resolve, use a different access modifier, or make the property non-abstract.

Example

The following sample generates CS0442:

C#

```
// CS0442.cs
public abstract class MyClass
{
    public abstract int AbstractProperty
    {
        get;
        private set;   // CS0442
        // Try this instead:
        // set;
    }
}
```

Compiler Error CS0443

Article • 09/15/2021

Syntax error, value expected

This error occurs when you reference an array without specifying a value for the array index.

Example

The following code generates CS0443.

C#

```
// CS0443.cs
using System;
class MyClass
{
    public static void Main()
    {
        int[,] x = new int[1,5];
        if (x[] == 5) {} // CS0443
        // if (x[0, 0] == 5) {}
    }
}
```

Compiler Error CS0445

Article • 09/15/2021

Cannot modify the result of an unboxing conversion

The result of an unboxing conversion is a temporary variable. The compiler prevents you from modifying such variables because any modification would go away when the temporary variable goes away. To fix this, declare a new value-type variable to store the intermediate expression, and assign the result of the unboxing conversion to that variable.

The following code generates CS0455.

C#

```
// CS0445.CS
class UnboxingTest
{
    public static void Main()
    {
        Point p;
        p.x = 1;
        p.y = 2;
        object obj = p;
        // The following line generates CS0445, because the result
        // of unboxing obj is a temporary variable.
        ((Point)obj).x = 2;

        // The following lines resolve the error.

        // Store the result of the unboxing conversion in p2.
        Point p2;
        p2 = (Point)obj;
        // Then you can modify the unboxed value.
        p2.x = 2;
    }
}

struct Point
{
    public int x, y;
}
```

Compiler Error CS0446

Article • 09/15/2021

Foreach cannot operate on a 'Method or Delegate'. Did you intend to invoke the 'Method or Delegate'?

This error is caused by specifying a method without parentheses or an anonymous method without parentheses in the part of the `foreach` statement where you would normally put a collection class. Note that it is valid, though unusual, to put a method call in that location, if the method returns a collection class.

Example

The following code will generate CS0446.

C#

```
// CS0446.cs
using System;
class Tester
{
    static void Main()
    {
        int[] intArray = new int[5];
        foreach (int i in M) { } // CS0446
    }
    static void M() { }
}
```

Compiler Error CS0449

08/05/2025

The `'class'`, `'struct'`, `'unmanaged'`, `'notnull'`, and `'default'` constraints cannot be combined or duplicated, and must be specified first in the constraints list.

The constraints on the type parameter of a generic type or method must occur in a specific order. You can apply at most one of the `class`, `struct`, `unmanaged`, `notnull`, or `default` constraints, and if you specify any of these constraints, it must be the first constraint specified for that type parameter. These are followed by interface constraints, and finally any constructor constraints. This error is caused by one of these primary constraints (`class`, `struct`, `unmanaged`, `notnull`, or `default`) not appearing first, or by attempting to combine multiple primary constraints. To resolve this error, reorder the constraint clauses or remove duplicate constraints.

> **Note**
>
> In a nullable context, the `class` constraint already implies `notnull`, so they cannot be combined. For comprehensive information about all constraint types and their rules, see Constraints on type parameters.

Example

The following sample generates CS0449.

C#

```
// CS0449.cs
// compile with: /target:library
public interface I {}  // Made public to avoid CS0703

public class C4
{
   public void F1<T>() where T : class, struct, I {}   // CS0449 - cannot combine
class and struct
   public void F2<T>() where T : I, struct {}   // CS0449 - struct must be first
   public void F3<T>() where T : I, class {}   // CS0449 - class must be first
   public void F4<T>() where T : class, notnull {}   // CS0449 - class already
implies notnull in nullable context
   public void F5<T>() where T : unmanaged, struct {}   // CS0449 - cannot combine
unmanaged and struct
   public void F6<T>() where T : I, unmanaged {}   // CS0449 - unmanaged must be
first
   public void F7<T>() where T : notnull, default {}   // CS0449 - cannot combine
notnull and default
```

```
    // OK
    public void F8<T>() where T : class {}
    public void F9<T>() where T : struct {}
    public void F10<T>() where T : unmanaged {}
    public void F11<T>() where T : notnull {}
    public void F12<T>() where T : I {}
}
```

Compiler Error CS0450

Article • 09/15/2021

'Type Parameter Name': cannot specify both a constraint class and the 'class' or 'struct' constraint

If a type parameter is constrained by the struct type constraint, it is logically contradictory for it to also be constrained by a specific class type, since struct and class are mutually exclusive categories. If a type parameter is constrained by a specific class type constraint, then it is by definition constrained by the class type constraint, and so specifying the class type constraint is redundant.

Example

C#

```
// CS0450.cs
// compile with: /t:library
public class GenericsErrors
{
    public class B { }
    public class G3<T> where T : struct, B { } // CS0450
// To resolve, use the following line instead:
// public class G3<T> where T : B { }
}
```

Compiler Error CS0451

Article • 09/15/2021

The 'new()' constraint cannot be used with the 'struct' constraint

When specifying constraints on the type of a generic, the `new()` constraint may only be used with class type constraints, interface type constraints, reference type constraints, and type parameter constraints, but not with value type constraints.

Example

The following sample generates CS0451.

C#

```
// CS0451.cs
using System;
public class C4
{
   public void F4<T>() where T : struct, new() {}   // CS0451
}

// OK
public class C5
{
   public void F5<T>() where T : struct {}
}

public class C6
{
   public void F6<T>() where T : new() {}
}
```

Compiler Error CS0452

Article • 09/15/2021

The type 'type name' must be a reference type in order to use it as parameter 'parameter name' in the generic type or method 'identifier of generic'

This error occurs when you pass a value type such as a `struct` or `int` as a parameter to a generic type or method that has a reference type constraint.

Example

The following code generates error CS0452.

C#

```
// CS0452.cs
using System;
public class BaseClass<S> where S : class { }
public class Derived1 : BaseClass<int> { } // CS0452
public class Derived2<S> : BaseClass<S> where S : struct { } // CS0452
```

See also

- Constraints on Type Parameters

Compiler Error CS0453

Article • 08/03/2023

The type 'Type Name' must be a non-nullable value type in order to use it as parameter 'Parameter Name' in the generic type or method 'Generic Identifier'

This error occurs when you use a non-value type argument in instantiating a generic type or method that has the **value** constraint on it. It can also occur when you use a nullable value type argument. See the last two lines of code in the following example.

Example

The following code generates this error.

C#

```
// CS0453.cs
using System;
public class HV<S> where S : struct { }

// CS0453: string is not a value type
public class H1 : HV<string> { }

// CS0453: H1 is a class, not a struct
public class H2 : HV<H1> { }

// CS0453: HV is based on a class, not a struct
public class H3<S> : HV<S> where S : class { }
public class H4<S> : HV<S> where S : struct { } // OK

// CS0453: HV accepts a nullable int type
public class H4 : HV<int?> { }

// CS0453: HV is based on Nullable type of int
public class H5 : HV<Nullable<Nullable<int>>> { }
```

See also

- Built-in reference types
- Value types
- Constraints on type parameters

Compiler Error CS0454

Article • 09/15/2021

Circular constraint dependency involving 'Type Parameter 1' and 'Type Parameter 2'

This error arises because at some point one type parameter refers to another, and the second refers back to the first. To fix this error, break the circular dependency by removing one of the constraints. Note that the circular constraint dependency can be indirect.

Example 1

The following code generates error CS0454.

C#

```
// CS0554
using System;
public class GenericsErrors
{
    public class G4<T> where T : T { } // CS0454
}
```

Example 2

The following example demonstrates a circular dependency between two type constraints.

C#

```
public class Gen<T,U> where T : U where U : T  // CS0454
{
}
```

Compiler Error CS0455

Article • 09/15/2021

Type parameter 'Type Parameter Name' inherits conflicting constraints 'Constraint Name 1' and 'Constraint Name 2'

Two common ways to get this error are to set up constraints so that the type parameter derives from two non-related classes, or so that it derives from a class type or reference type constraint and a `struct` type or value type constraint. To resolve this error, remove the conflict from your inheritance hierarchy.

Example

The following code generates error CS0455.

C#

```
// CS0455.cs
using System;

public class GenericsErrors {
    public class B { }
    public class B2 { }
    public class G6<T> where T : B { public class N<U> where U : B2, T { } }
// CS0455
}
```

Compiler Error CS0456

Article • 09/15/2021

Type parameter 'Type Parameter Name 1' has the 'struct' constraint so 'Type Parameter Name 1' cannot be used as a constraint for 'Type Parameter Name 2'

Value type constraints are implicitly sealed so those constraints cannot be used as constraints on a second type parameter. This is because value types cannot be overridden. To resolve this error, put a value type constraint directly on the second type parameter, instead of doing so indirectly by means of the first type parameter.

Example

The following sample generates CS0456.

C#

```
// CS0456.cs
// compile with: /target:library
public class GenericsErrors
{
   public class G5<T> where T : struct
   {
      public class N<U> where U : T {}   // CS0456
      public class N2<U> where U : struct {}   // OK
   }
}
```

Compiler Error CS0462

Article • 09/15/2021

The inherited members 'member1' and 'member2' have the same signature in type 'type', so they cannot be overridden

This error arises with the introduction of generics. Normally, you cannot have two versions of a method in a class with the same signature. But with generics, you can specify a generic method that might duplicate another method if it is instantiated with a particular type.

Example

When `C<int>` is instantiated, two versions of the method `F` are created with the same signature, so the override in class `D` cannot decide which one to apply the override to.

The following sample generates CS0462.

C#

```
// CS0462.cs
// compile with: /target:library
class C<T>
{
   public virtual void F(T t) {}
   public virtual void F(int t) {}
}

class D : C<int>
{
   public override void F(int t) {}   // CS0462
}
```

Compiler Error CS0463

Article • 09/08/2022

Evaluation of the decimal constant expression failed with error: 'error'

This error occurs when a constant decimal expression overflows at compile time.

Typically you get overflow errors at run time. In this case, you defined the constant expression in such a way that the compiler could evaluate the result and know that an overflow would happen.

Example

The following code generates error CS0463.

C#

```
// CS0463.cs
using System;
class MyClass
{
    public static void Main()
    {
        const decimal myDec = 7900000000000000000000000000.0m +
7900000000000000000000000000.0m; // CS0463
        Console.WriteLine(myDec.ToString());
    }
}
```

Compiler Error CS0468

Article • 09/15/2021

Ambiguity between type 'type1' and type 'type2'

This error is generated when there are two types in the assembly being compiled that have the same fully qualified name. This could occur if they are both in added modules or one is in an added module and one in source.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Resolve errors and warnings related to members that implement an interface

This article covers the following compiler errors:

- CS0071: *An explicit interface implementation of an event must use event accessor syntax.*
- CS0106: *The modifier is not valid for this item.*
- CS0277: *Member does not implement interface member because it is not public.*
- CS0425: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- CS0460: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*
- CS0470: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- CS0473: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- CS0531: *Interface members cannot have a definition.*
- CS0535: *Member does not implement interface member.*
- CS0538: *Member in explicit interface declaration is not an interface.*
- CS0539: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- CS0540: *Containing type does not implement interface member.*
- CS0541: *Explicit interface declaration can only be declared in a class, record, struct or interface.*
- CS0550: *Member adds an accessor not found in interface member.*
- CS0551: *Explicit interface implementation is missing an accessor.*
- CS0630: *Member cannot implement interface member because it has an __arglist parameter.*
- CS0686: *Accessor cannot implement interface member. Use an explicit interface implementation.*
- CS0736: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- CS0737: *Member does not implement interface member. It cannot implement an interface member because it is not public.*
- CS0738: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- CS8705: *Interface member does not have a most specific implementation. Neither member is most specific.*

- CS8854: *Member does not implement interface member.*
- CS9333: *Parameter type must match implemented member.*
- CS9334: *Return type must match implemented member.*

Interface declaration and syntax

The following errors relate to proper syntax and structure when declaring explicit interface implementations:

- **CS0071**: *An explicit interface implementation of an event must use event accessor syntax.*
- **CS0106**: *The modifier is not valid for this item.*
- **CS0531**: *Interface members cannot have a definition.*
- **CS0538**: *Member in explicit interface declaration is not an interface.*
- **CS0541**: *Explicit interface declaration can only be declared in a class, record, struct or interface.*

You can correct these errors using the following techniques:

- You must manually provide `add` and `remove` event accessors when explicitly implementing an interface event (**CS0071**). The compiler doesn't automatically generate these accessors for explicit interface implementations, so you must define them explicitly to specify how the event is stored and managed.
- Remove the `public` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations are implicitly public when accessed through the interface type, making the `public` keyword redundant and not allowed in this context.
- Remove the `abstract` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations provide the actual implementation and can't be marked as abstract because they can't be overridden in derived classes.
- Remove the method body from interface member declarations, or move the implementation to a class or struct that implements the interface (**CS0531**). Before C# 8.0, interface members can't contain implementations; starting with C# 8.0, you can provide default interface methods using specific syntax.
- Verify that the type specified in the explicit interface declaration is an actual interface type (**CS0538**). Only interface types can be used in explicit interface implementation syntax; attempting to use a class or other non-interface type violates the explicit implementation rules.
- Move explicit interface declarations into a class or struct that declares the interface in its base list (**CS0541**). Explicit interface implementations must appear within the body of a class or struct type and can't be declared at the namespace level or in other contexts.

For more information, see Interfaces, Explicit Interface Implementation, and How to implement interface events.

Return types and signatures

The following errors occur when the implementing method's signature doesn't match the interface member declaration:

- **CS0738**: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- **CS8854**: *Member does not implement interface member.*
- **CS9333**: *Parameter type must match implemented member.*
- **CS9334**: *Return type must match implemented member.*

You can correct these errors using the following techniques:

- Change the return type of the implementing method to exactly match the return type declared in the interface member (**CS0738**, **CS9334**). The signature of the implementation must match the interface declaration precisely because the method signature is part of the contract that determines which interface member is being implemented.
- Ensure that parameter types in the implementing method exactly match the parameter types declared in the interface member (**CS9333**). Each parameter must have the identical type in the same position as specified in the interface declaration, as parameter types are fundamental components of the method signature that the compiler uses to match implementations to interface members.
- Add an `init` accessor to the implementing property when the interface property declares an `init` setter (**CS8854**). The `init` keyword allows property initialization during object construction while preventing modification afterward, and the implementing property must provide this same initialization-only behavior to satisfy the interface contract.

For more information, see Interfaces, Properties, and Init-only setters.

Missing or incomplete implementations

The following errors occur when a class fails to fully implement an interface or implements members that don't match the interface contract:

- **CS0535**: *Member does not implement interface member.*
- **CS0550**: *Member adds an accessor not found in interface member.*
- **CS0551**: *Explicit interface implementation is missing an accessor.*

You can correct these errors using the following techniques:

- Provide an implementation for every member declared in the interface, or declare the type as `abstract` (**CS0535**). Each member must be implemented to satisfy the interface requirements.
- Remove any accessors from the implementing property that aren't declared in the interface property (**CS0550**). The implementing property can only include the accessors explicitly declared in the interface definition, ensuring that the implementation doesn't add functionality beyond what the interface contract specifies.
- Add all required accessors to the explicit interface implementation to match the interface declaration (**CS0551**). Each accessor declared in the interface must have a corresponding accessor in the implementation with matching signatures, as the implementation must fulfill the complete accessor contract defined by the interface.

For more information, see Interfaces and Properties.

Member matching and resolution

The following errors occur when attempting to implement interface members that don't exist in the interface or when the containing type doesn't declare the interface:

- **CS0539**: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- **CS0540**: *Containing type does not implement interface member.*

You can correct these errors using the following techniques:

- Verify that the member name and signature in the explicit interface implementation exactly match a member declared in the interface, or remove the incorrect implementation (**CS0539**). The member you're attempting to implement must actually exist in the interface definition with matching name, return type, and parameter types, as explicit interface implementation requires precise correspondence with the interface contract.
- Add the interface to the class's or struct's base list, or remove the explicit interface implementation (**CS0540**). A type can only explicitly implement members of interfaces that it declares in its inheritance list, so the implementing type must establish the interface relationship before it can provide explicit implementations.

For more information, see Interfaces and Explicit Interface Implementation.

Generic type constraints

The following errors occur when implementing generic interface methods with type parameter constraints:

- **CS0425**: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- **CS0460**: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*

You can correct these errors using the following techniques:

- Ensure the `where` clause in the implementing method is identical to the interface method declaration, or matches the semantic meaning of the constraints (**CS0425**). The type parameter constraints in the implementation must match those defined in the interface or base method.
- Remove explicit constraint declarations from override and explicit interface implementation methods (**CS0460**). The override method inherits its constraints automatically from the base or interface method, so redeclaring them is redundant and not permitted except for specific cases allowed in C# 8 and later.
- Apply the `default` constraint to resolve ambiguities with nullable reference types in override and explicit interface implementations when using C# 9 or later (**CS0460**). This exception to the constraint inheritance rule allows you to explicitly specify the default constraint to disambiguate nullable annotation contexts.
- Explicitly specify `where T : class` or `where T : struct` constraints on override and explicit interface implementation methods when using C# 8 or later to enable nullable reference type annotations (**CS0460**). These specific constraints are permitted to support nullable reference type analysis on type parameters that are constrained to reference or value types.

For more information, see Constraints on type parameters, Interfaces, and Nullable reference types.

Method visibility and modifiers

The following errors occur when implementing interface methods with incorrect accessibility or modifiers:

- **CS0736**: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- **CS0737**: *Member does not implement interface member. It cannot implement an interface member because it is not public.*

You can correct these errors using the following techniques:

- Remove the `static` modifier from the method declaration that implements the interface member (**CS0736**). Before C# 10, interface members are instance members, not static members.
- Add the `public` access modifier to the method that implements the interface member (**CS0737**). All interface members are implicitly `public` because interfaces define a contract for public behavior, so the implementing method must also have public accessibility to be accessible through the interface reference.

For more information, see Interfaces and Access Modifiers.

Accessor implementation and conflicts

The following errors occur when implementing interface properties or events with accessor methods that have visibility issues or naming conflicts:

- **CS0277**: *Member does not implement interface member because it is not public.*
- **CS0470**: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- **CS0686**: *Accessor cannot implement interface member. Use an explicit interface implementation.*

You can correct these errors using the following techniques:

- Remove any access modifiers from property accessors that restrict visibility to less visible than `public`, or add the `public` modifier if it's missing (**CS0277**). All interface members are implicitly `public`, so the implementing accessor must also have public accessibility to satisfy the interface contract and be accessible through the interface type.
- Replace methods with accessor-like names (such as `get_PropertyName`) with proper property syntax using explicit interface implementation (**CS0470**). The compiler generates accessor methods internally for properties, and attempting to manually create methods with these reserved names conflicts with the property implementation mechanism.
- Use explicit interface implementation syntax to resolve naming conflicts when the interface contains method names that match the auto-generated accessor methods (**CS0686**). The compiler automatically generates methods like `get_Property` and `set_Property` for properties, and `add_Event` and `remove_Event` for events, so if an interface declares methods with these exact names, explicit implementation is required to disambiguate between the interface method and the compiler-generated accessor.

For more information, see Interfaces, Properties, and Events.

Ambiguous and conflicting implementations

The following errors occur when the compiler can't determine which interface implementation to use:

- **CS0473**: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- **CS8705**: *Interface member 'member' does not have a most specific implementation. Neither is most specific.*

You can correct these errors using the following techniques:

- Eliminate the explicit interface implementation and instead use a single implicit public implementation for both interface methods (**CS0473**). When a generic method acquires the same signature as a non-generic method (such as when implementing `ITest<int>` where both `TestMethod(int)` and `TestMethod(T)` become identical), the common language infrastructure metadata system can't unambiguously determine which interface member binds to which implementation slot, so using implicit implementation allows the single method to satisfy both interface requirements.
- Provide an explicit implementation in the implementing class or struct that resolves the ambiguity between multiple default implementations (**CS8705**). This error typically occurs with diamond inheritance patterns where a class implements multiple interfaces that each provide default implementations for the same member. The compiler needs you to explicitly specify which implementation to use, or provide your own implementation.
- Restructure the interface hierarchy to avoid diamond inheritance conflicts where multiple interfaces provide default implementations for the same member (**CS8705**). By redesigning the interface relationships or consolidating the default implementations into a single interface, you can eliminate the ambiguity that prevents the compiler from determining the most specific implementation.

For more information, see Interfaces and Default Interface Methods.

Special implementation restrictions

The following error occurs when using special parameter types that aren't compatible with interface implementation:

- **CS0630**: *Member cannot implement interface member because it has an __arglist parameter.*

You can correct this error using the following techniques:

- Remove the `__arglist` parameter from the implementing method (**CS0630**). The `__arglist` keyword allows methods to accept variable numbers of arguments in an unmanaged way, but this feature is incompatible with interface implementation because interface contracts require predictable, type-safe signatures that can be verified at compile time.
- Replace the `__arglist` parameter with a `params` array parameter for variable-length argument lists (**CS0630**). Unlike `__arglist`, the `params` keyword provides a type-safe mechanism for accepting variable numbers of arguments that is fully compatible with interface implementation and maintains the compile-time type safety that interfaces require.

For more information, see Interfaces and params keyword.

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

---

Last updated on 11/14/2025

Compiler Error CS0471

Article • 09/15/2021

The method 'name' is not a generic method. If you intended an expression list, use parentheses around the < expression.

This error is generated when your code contains an expression list without parentheses.

Example

The following example generates CS0471:

C#

```
// CS0471.cs
// compile with: /t:library
class Test
{
    public void F(bool x, bool y) {}
    public void F1()
    {
        int a = 1, b = 2, c = 3;
        F(a<b, c>(3));    // CS0471
        // To resolve, try the following instead:
        // F((a<b), c>(3));
    }
}
```

> **Note**
>
> This compiler error is no longer used in Roslyn. The previous example should compile successfully.

Compiler Error CS0500

Article • 09/15/2021

'class member' cannot declare a body because it is marked abstract

An abstract method cannot contain its implementation.

The following sample generates CS0500:

C#

```
// CS0500.cs
namespace x
{
   abstract public class clx
   {
      abstract public void f(){}   // CS0500
      // try the following line instead
      // abstract public void f();
   }

   public class cly
   {
      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0502

Article • 09/15/2021

'member' cannot be both abstract and sealed

A class member cannot be both abstract and sealed.

The following sample generates CS0502:

C#

```
// CS0502.cs
public class B
{
   abstract public void F();
}

public class C : B
{
   abstract sealed override public void F()   // CS0502
   {
   }
}

public class CMain
{
    public static void Main()
    { }
}
```

Compiler Error CS0503

Article • 09/15/2021

The abstract method 'method' cannot be marked virtual

It is redundant to mark a member method as both abstract and virtual because **abstract** implies **virtual**.

The following sample generates CS0503:

C#

```
// CS0503.cs
namespace x
{
   abstract public class clx
   {
      abstract virtual public void f();   // CS0503
   }
}
```

Compiler Error CS0504

Article • 09/15/2021

The constant 'variable' cannot be marked static

If a variable is const, it is also static. If you want a **const** and **static** variable, just declare that variable as **const**; if all you want is a **static** variable, just mark it **static**.

The following sample generates CS0504:

C#

```
// CS0504.cs
namespace x
{
   abstract public class clx
   {
      static const int i = 0;   // CS0504, cannot be both static and const
      abstract public void f();
   }
}
```

Compiler Error CS0505

Article • 09/15/2021

'member1': cannot override because 'member2' is not a function

A class declaration attempted to override a non-method in a base class. Overrides must match the member type. If a method with the same name as a method in a base class is desired, use new (and not override) on the method declaration in the base class.

The following sample generates CS0505:

C#

```
// CS0505.cs
// compile with: /target:library
public class clx
{
   public int i;
}

public class cly : clx
{
   public override int i() { return 0; }   // CS0505
}
```

Compiler Error CS0506

Article • 09/15/2021

'function1' : cannot override inherited member 'function2' because it is not marked "virtual", "abstract", or "override"

A method was overridden that was not explicitly marked as **virtual**, **abstract**, or `override`.

The following sample generates CS0506:

C#

```
// CS0506.cs
namespace MyNameSpace
{
   abstract public class ClassX
   {
      public int i = 0;

      public int f()
      {
         return 0;
      }
      // Try the following definition for f() instead:
      // abstract public int f();
   }

   public class ClassY : ClassX
   {
      public override int f()   // CS0506
      {
         return 0;
      }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0507

Article • 05/03/2023

'function1' : cannot change access modifiers when overriding 'access' inherited member 'function2'

An attempt was made to change the access specification in a method override.

Example 1

The following sample generates CS0507.

C#

```
// CS0507.cs
abstract public class clx
{
   virtual protected void f() {}
}

public class cly : clx
{
   public override void f() {}   // CS0507
   public static void Main() {}
}
```

Example 2

CS0507 can also occur if a class attempts to override a method marked as `protected internal` defined in referenced metadata. In this situation, the overriding method should be marked as `protected`.

C#

```
// CS0507_b.cs
// compile with: /target:library
abstract public class clx
{
   virtual protected internal void f() {}
}
```

The following sample generates CS0507.

C#

```
// CS0507_c.cs
// compile with: /reference:cs0507_b.dll
public class cly : clx
{
   protected internal override void f() {}   // CS0507
   // try the following line instead
   // protected override void f() {}   // OK

   public static void Main() {}
}
```

Compiler Error CS0508

Article • 09/15/2021

'Type 1': return type must be 'Type 2' to match overridden member 'Member Name'

An attempt was made to change the return type in a method override. To resolve this error, make sure both methods declare the same return type.

Example

The following sample generates CS0508.

C#

```
// CS0508.cs
// compile with: /target:library
abstract public class Clx
{
   public int i = 0;
   // Return type is int.
   abstract public int F();
}

public class Cly : Clx
{
   public override double F()
   {
      return 0.0;   // CS0508
   }
}
```

Compiler Error CS0509

Article • 09/15/2021

'class1' : cannot derive from sealed type 'class2'

A sealed class cannot act as a base class. Structs are sealed by default.

The following sample generates CS0509:

C#

```
// CS0509.cs
// compile with: /target:library
sealed public class clx {}
public class cly : clx {}   // CS0509
```

Compiler Error CS0513

Article • 09/15/2021

'function' is abstract but it is contained in nonabstract class 'class'

A method cannot be an abstract member of a nonabstract class.

The following sample generates CS0513:

C#

```
// CS0513.cs
namespace x
{
   public class clx
   {
      abstract public void f();   // CS0513, abstract member of nonabstract
class
   }
}
```

Compiler Error CS0518

08/07/2025

Predefined type 'type' is not defined or imported

> **Note**
>
> The resolution for this error depends on whether you're using a modern SDK-style project (`.csproj` files that start with `<Project Sdk="Microsoft.NET.Sdk">`) or legacy project formats. SDK-style projects manage runtime references automatically through the `<TargetFramework>` property.

The main cause for this problem is that the project cannot access the predefined types from the .NET runtime library. In modern SDK-style projects, this is typically due to an incorrect or missing `<TargetFramework>` specification. In legacy projects, this issue is caused by not importing mscorlib.dll, which defines the entire System namespace. This can be caused by one of the following:

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

- The **NoStandardLib** option from the command line compiler has been specified. The **NoStandardLib** option prevents the import of mscorlib.dll. Use this option if you want to define or create a user-specific System namespace.
- An incorrect mscorlib.dll is referenced.
- A corrupt Visual Studio .NET or .NET Framework common language runtime installation exists.
- Residual components from an earlier installation that are incompatible with the latest installation remain.

To resolve this problem, take one of the following actions:

- Do not specify the /nostdlib option from the command line compiler.

- For modern SDK-style projects, ensure the project targets the correct .NET runtime. In your `.csproj` file, verify the `<TargetFramework>` property specifies the intended runtime:

  ```xml
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>
  ```

  For multi-targeting projects, use `<TargetFrameworks>` (plural):

  ```xml
  <PropertyGroup>
    <TargetFrameworks>net8.0;net48</TargetFrameworks>
  </PropertyGroup>
  ```

- For legacy project formats, make sure that the project refers to the correct mscorlib.dll.
- Reinstall the .NET Framework common language runtime (if the previous solutions do not solve the problem).
- Reload the project in Visual Studio.
- Close Visual Studio, delete the `obj` and `bin` folders from your project directory, then reopen Visual Studio and rebuild the project.

Compiler Error CS0520

Article • 09/15/2021

Predefined type 'name' is declared incorrectly

One or more files needed by the compiler were not found. Reinstall the .NET runtime.

Compiler Error CS0523

Article • 12/21/2023

Struct member 'struct2 field' of type 'struct1' causes a cycle in the struct layout

The definitions of one or more structs include recursive references that form a cycle. This limitation only applies to structs, since structs are value types. To create recursive references, declare your types as classes, which are reference types.

Example 1

The following sample shows how a self referential type can cause CS0523:

C#

```
// CS0523.cs
// compile with: /target:library
struct SelfReferentialStruct
{
    public SelfReferentialStruct other;   // CS0523
}

class SelfReferentialClass
{
    public SelfReferentialClass other;   // OK
}
```

When a self referential struct type is made, it contains a copy of the same type as a member. However, that member then has another copy, which continues recursively. As a result of the cycle, the size of the type cannot be determined and CS0523 is emitted.

Example 2

The following sample shows how a type reference cycle can cause CS0523:

C#

```
// CS0523b.cs
// compile with: /target:library
struct ReferenceCycleStruct1
{
    public ReferenceCycleStruct2 other;   // CS0523
}

struct ReferenceCycleStruct2
```

```
{
    public ReferenceCycleStruct3 other;   // CS0523
}

struct ReferenceCycleStruct3
{
    public ReferenceCycleStruct1 other;   // CS0523
}
```

To resolve the errors above, you can adjust the references such that a cycle is no longer formed, or convert at least one of the struct types to a class. Similar to the previous example, `ReferenceCycleStruct1` contains a `ReferenceCycleStruct2`, and that contains a `ReferenceCycleStruct3`, which eventually contains `ReferenceCycleStruct1` again.

Compiler Error CS0524

08/05/2025

'type' : interfaces cannot declare types

Prior to C# 8.0, an interface could not contain user-defined types. Starting with C# 8.0, interfaces can declare nested types as part of the default interface members feature.

In modern C# (C# 8.0 and later), this error is no longer generated for nested types in interfaces. The compiler now allows interfaces to contain nested classes, structs, interfaces, enums, and delegates.

Historical example

The following sample generates CS0524 in C# versions prior to 8.0, but is valid in modern C# versions:

C#

```
// This code is valid in C# 8.0 and later
public interface IExample
{
    public class NestedClass   // Valid since C# 8.0
    {
    }
}
```

See also

- Interfaces
- Nested types
- What's new in C# version history

Compiler Error CS0525

06/10/2025

Interfaces cannot contain instance fields

An interface can contain methods, properties, and fields.

The following sample generates CS0525:

```
C#
```

```
// CS0525.cs
namespace x
{
   public interface clx
   {
      public int i;   // CS0525
   }
}
```

Only classes can contain instance data. Interfaces describe a contract. These articles help you learn their key differences and use cases.

Compiler Error CS0527

Article • 09/15/2021

Type 'type' in interface list is not an interface

It is possible for a struct or interface to inherit from another interface but not from any other type.

The following sample generates CS0527:

```
C#
```

```
// CS0527.cs
// compile with: /target:library
public struct clx : int {}   // CS0527 int not an interface
```

Compiler Error CS0528

Article • 09/15/2021

'interface' is already listed in interface list

An interface-inheritance list includes a duplicate. An interface can only be specified once in the inheritance list.

The following sample generates CS0528:

C#

```
// CS0528.cs
namespace x
{
   public interface a
   {
   }

   public class b : a, a   // CS0528
   {
      public void Main()
      {
      }
   }
}
```

Compiler Error CS0529

Article • 09/15/2021

Inherited interface 'interface1' causes a cycle in the interface hierarchy of 'interface2'

The inheritance list for an interface includes a direct or indirect reference to itself. An interface cannot inherit from itself.

The following sample generates CS0529:

C#

```
// CS0529.cs
namespace x
{
   public interface a
   {
   }

   public interface b : a, c
   {
   }

   public interface c : b   // CS0529, b inherits from c
   {
   }
}
```

Compiler Error CS0533

Article • 09/15/2021

'derived-class member' hides inherited abstract member 'base-class member'

A base class method is hidden. Check the syntax of your declaration to see if it is correct.

For more information, see base.

The following sample generates CS0533:

C#

```
// CS0533.cs
namespace x
{
   abstract public class a
   {
      abstract public void f();
   }

   abstract public class b : a
   {
      new abstract public void f();   // CS0533
      // try the following lines instead
      // override public void f()
      // {
      // }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0534

Article • 09/15/2021

'function1' does not implement inherited abstract member 'function2'

A class is required to implement all the abstract members in the base class, unless the class is also abstract.

The following sample generates CS0534:

C#

```
// CS0534.cs
namespace x
{
   abstract public class clx
   {
      public abstract void f();
   }

   public class cly : clx   // CS0534, no override for clx::f
   {
      // uncomment the following sample override to resolve CS0534
      // override public void f()
      // {
      // }

      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0537

Article • 09/15/2021

The class System.Object cannot have a base class or implement an interface

CS0537 occurs when rebuilding the System class libraries, and where Object derives from another class. You are very unlikely to encounter this error. If you do, do not derive Object from a class or interface: it is the root of the entire .NET class hierarchy, and as such, does not derive from anything else.

Resolve errors and warnings related to members that implement an interface

This article covers the following compiler errors:

- CS0071: *An explicit interface implementation of an event must use event accessor syntax.*
- CS0106: *The modifier is not valid for this item.*
- CS0277: *Member does not implement interface member because it is not public.*
- CS0425: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- CS0460: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*
- CS0470: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- CS0473: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- CS0531: *Interface members cannot have a definition.*
- CS0535: *Member does not implement interface member.*
- CS0538: *Member in explicit interface declaration is not an interface.*
- CS0539: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- CS0540: *Containing type does not implement interface member.*
- CS0541: *Explicit interface declaration can only be declared in a class, record, struct or interface.*
- CS0550: *Member adds an accessor not found in interface member.*
- CS0551: *Explicit interface implementation is missing an accessor.*
- CS0630: *Member cannot implement interface member because it has an __arglist parameter.*
- CS0686: *Accessor cannot implement interface member. Use an explicit interface implementation.*
- CS0736: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- CS0737: *Member does not implement interface member. It cannot implement an interface member because it is not public.*
- CS0738: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- CS8705: *Interface member does not have a most specific implementation. Neither member is most specific.*

- CS8854: *Member does not implement interface member.*
- CS9333: *Parameter type must match implemented member.*
- CS9334: *Return type must match implemented member.*

Interface declaration and syntax

The following errors relate to proper syntax and structure when declaring explicit interface implementations:

- **CS0071**: *An explicit interface implementation of an event must use event accessor syntax.*
- **CS0106**: *The modifier is not valid for this item.*
- **CS0531**: *Interface members cannot have a definition.*
- **CS0538**: *Member in explicit interface declaration is not an interface.*
- **CS0541**: *Explicit interface declaration can only be declared in a class, record, struct or interface.*

You can correct these errors using the following techniques:

- You must manually provide `add` and `remove` event accessors when explicitly implementing an interface event (**CS0071**). The compiler doesn't automatically generate these accessors for explicit interface implementations, so you must define them explicitly to specify how the event is stored and managed.
- Remove the `public` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations are implicitly public when accessed through the interface type, making the `public` keyword redundant and not allowed in this context.
- Remove the `abstract` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations provide the actual implementation and can't be marked as abstract because they can't be overridden in derived classes.
- Remove the method body from interface member declarations, or move the implementation to a class or struct that implements the interface (**CS0531**). Before C# 8.0, interface members can't contain implementations; starting with C# 8.0, you can provide default interface methods using specific syntax.
- Verify that the type specified in the explicit interface declaration is an actual interface type (**CS0538**). Only interface types can be used in explicit interface implementation syntax; attempting to use a class or other non-interface type violates the explicit implementation rules.
- Move explicit interface declarations into a class or struct that declares the interface in its base list (**CS0541**). Explicit interface implementations must appear within the body of a class or struct type and can't be declared at the namespace level or in other contexts.

For more information, see Interfaces, Explicit Interface Implementation, and How to implement interface events.

Return types and signatures

The following errors occur when the implementing method's signature doesn't match the interface member declaration:

- **CS0738**: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- **CS8854**: *Member does not implement interface member.*
- **CS9333**: *Parameter type must match implemented member.*
- **CS9334**: *Return type must match implemented member.*

You can correct these errors using the following techniques:

- Change the return type of the implementing method to exactly match the return type declared in the interface member (**CS0738**, **CS9334**). The signature of the implementation must match the interface declaration precisely because the method signature is part of the contract that determines which interface member is being implemented.
- Ensure that parameter types in the implementing method exactly match the parameter types declared in the interface member (**CS9333**). Each parameter must have the identical type in the same position as specified in the interface declaration, as parameter types are fundamental components of the method signature that the compiler uses to match implementations to interface members.
- Add an `init` accessor to the implementing property when the interface property declares an `init` setter (**CS8854**). The `init` keyword allows property initialization during object construction while preventing modification afterward, and the implementing property must provide this same initialization-only behavior to satisfy the interface contract.

For more information, see Interfaces, Properties, and Init-only setters.

Missing or incomplete implementations

The following errors occur when a class fails to fully implement an interface or implements members that don't match the interface contract:

- **CS0535**: *Member does not implement interface member.*
- **CS0550**: *Member adds an accessor not found in interface member.*
- **CS0551**: *Explicit interface implementation is missing an accessor.*

You can correct these errors using the following techniques:

- Provide an implementation for every member declared in the interface, or declare the type as `abstract` (**CS0535**). Each member must be implemented to satisfy the interface requirements.
- Remove any accessors from the implementing property that aren't declared in the interface property (**CS0550**). The implementing property can only include the accessors explicitly declared in the interface definition, ensuring that the implementation doesn't add functionality beyond what the interface contract specifies.
- Add all required accessors to the explicit interface implementation to match the interface declaration (**CS0551**). Each accessor declared in the interface must have a corresponding accessor in the implementation with matching signatures, as the implementation must fulfill the complete accessor contract defined by the interface.

For more information, see Interfaces and Properties.

Member matching and resolution

The following errors occur when attempting to implement interface members that don't exist in the interface or when the containing type doesn't declare the interface:

- **CS0539**: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- **CS0540**: *Containing type does not implement interface member.*

You can correct these errors using the following techniques:

- Verify that the member name and signature in the explicit interface implementation exactly match a member declared in the interface, or remove the incorrect implementation (**CS0539**). The member you're attempting to implement must actually exist in the interface definition with matching name, return type, and parameter types, as explicit interface implementation requires precise correspondence with the interface contract.
- Add the interface to the class's or struct's base list, or remove the explicit interface implementation (**CS0540**). A type can only explicitly implement members of interfaces that it declares in its inheritance list, so the implementing type must establish the interface relationship before it can provide explicit implementations.

For more information, see Interfaces and Explicit Interface Implementation.

Generic type constraints

The following errors occur when implementing generic interface methods with type parameter constraints:

- **CS0425**: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- **CS0460**: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*

You can correct these errors using the following techniques:

- Ensure the `where` clause in the implementing method is identical to the interface method declaration, or matches the semantic meaning of the constraints (**CS0425**). The type parameter constraints in the implementation must match those defined in the interface or base method.
- Remove explicit constraint declarations from override and explicit interface implementation methods (**CS0460**). The override method inherits its constraints automatically from the base or interface method, so redeclaring them is redundant and not permitted except for specific cases allowed in C# 8 and later.
- Apply the `default` constraint to resolve ambiguities with nullable reference types in override and explicit interface implementations when using C# 9 or later (**CS0460**). This exception to the constraint inheritance rule allows you to explicitly specify the default constraint to disambiguate nullable annotation contexts.
- Explicitly specify `where T : class` or `where T : struct` constraints on override and explicit interface implementation methods when using C# 8 or later to enable nullable reference type annotations (**CS0460**). These specific constraints are permitted to support nullable reference type analysis on type parameters that are constrained to reference or value types.

For more information, see Constraints on type parameters, Interfaces, and Nullable reference types.

Method visibility and modifiers

The following errors occur when implementing interface methods with incorrect accessibility or modifiers:

- **CS0736**: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- **CS0737**: *Member does not implement interface member. It cannot implement an interface member because it is not public.*

You can correct these errors using the following techniques:

- Remove the `static` modifier from the method declaration that implements the interface member (**CS0736**). Before C# 10, interface members are instance members, not static members.
- Add the `public` access modifier to the method that implements the interface member (**CS0737**). All interface members are implicitly `public` because interfaces define a contract for public behavior, so the implementing method must also have public accessibility to be accessible through the interface reference.

For more information, see Interfaces and Access Modifiers.

Accessor implementation and conflicts

The following errors occur when implementing interface properties or events with accessor methods that have visibility issues or naming conflicts:

- **CS0277**: *Member does not implement interface member because it is not public.*
- **CS0470**: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- **CS0686**: *Accessor cannot implement interface member. Use an explicit interface implementation.*

You can correct these errors using the following techniques:

- Remove any access modifiers from property accessors that restrict visibility to less visible than `public`, or add the `public` modifier if it's missing (**CS0277**). All interface members are implicitly `public`, so the implementing accessor must also have public accessibility to satisfy the interface contract and be accessible through the interface type.
- Replace methods with accessor-like names (such as `get_PropertyName`) with proper property syntax using explicit interface implementation (**CS0470**). The compiler generates accessor methods internally for properties, and attempting to manually create methods with these reserved names conflicts with the property implementation mechanism.
- Use explicit interface implementation syntax to resolve naming conflicts when the interface contains method names that match the auto-generated accessor methods (**CS0686**). The compiler automatically generates methods like `get_Property` and `set_Property` for properties, and `add_Event` and `remove_Event` for events, so if an interface declares methods with these exact names, explicit implementation is required to disambiguate between the interface method and the compiler-generated accessor.

For more information, see Interfaces, Properties, and Events.

Ambiguous and conflicting implementations

The following errors occur when the compiler can't determine which interface implementation to use:

- **CS0473**: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- **CS8705**: *Interface member 'member' does not have a most specific implementation. Neither is most specific.*

You can correct these errors using the following techniques:

- Eliminate the explicit interface implementation and instead use a single implicit public implementation for both interface methods (**CS0473**). When a generic method acquires the same signature as a non-generic method (such as when implementing `ITest<int>` where both `TestMethod(int)` and `TestMethod(T)` become identical), the common language infrastructure metadata system can't unambiguously determine which interface member binds to which implementation slot, so using implicit implementation allows the single method to satisfy both interface requirements.
- Provide an explicit implementation in the implementing class or struct that resolves the ambiguity between multiple default implementations (**CS8705**). This error typically occurs with diamond inheritance patterns where a class implements multiple interfaces that each provide default implementations for the same member. The compiler needs you to explicitly specify which implementation to use, or provide your own implementation.
- Restructure the interface hierarchy to avoid diamond inheritance conflicts where multiple interfaces provide default implementations for the same member (**CS8705**). By redesigning the interface relationships or consolidating the default implementations into a single interface, you can eliminate the ambiguity that prevents the compiler from determining the most specific implementation.

For more information, see Interfaces and Default Interface Methods.

Special implementation restrictions

The following error occurs when using special parameter types that aren't compatible with interface implementation:

- **CS0630**: *Member cannot implement interface member because it has an __arglist parameter.*

You can correct this error using the following techniques:

- Remove the `__arglist` parameter from the implementing method (**CS0630**). The `__arglist` keyword allows methods to accept variable numbers of arguments in an unmanaged way, but this feature is incompatible with interface implementation because interface contracts require predictable, type-safe signatures that can be verified at compile time.
- Replace the `__arglist` parameter with a `params` array parameter for variable-length argument lists (**CS0630**). Unlike `__arglist`, the `params` keyword provides a type-safe mechanism for accepting variable numbers of arguments that is fully compatible with interface implementation and maintains the compile-time type safety that interfaces require.

For more information, see Interfaces and params keyword.

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

---

Last updated on 11/14/2025

Resolve errors and warnings related to members that implement an interface

This article covers the following compiler errors:

- CS0071: *An explicit interface implementation of an event must use event accessor syntax.*
- CS0106: *The modifier is not valid for this item.*
- CS0277: *Member does not implement interface member because it is not public.*
- CS0425: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- CS0460: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*
- CS0470: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- CS0473: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- CS0531: *Interface members cannot have a definition.*
- CS0535: *Member does not implement interface member.*
- CS0538: *Member in explicit interface declaration is not an interface.*
- CS0539: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- CS0540: *Containing type does not implement interface member.*
- CS0541: *Explicit interface declaration can only be declared in a class, record, struct or interface.*
- CS0550: *Member adds an accessor not found in interface member.*
- CS0551: *Explicit interface implementation is missing an accessor.*
- CS0630: *Member cannot implement interface member because it has an __arglist parameter.*
- CS0686: *Accessor cannot implement interface member. Use an explicit interface implementation.*
- CS0736: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- CS0737: *Member does not implement interface member. It cannot implement an interface member because it is not public.*
- CS0738: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- CS8705: *Interface member does not have a most specific implementation. Neither member is most specific.*

- CS8854: *Member does not implement interface member.*
- CS9333: *Parameter type must match implemented member.*
- CS9334: *Return type must match implemented member.*

Interface declaration and syntax

The following errors relate to proper syntax and structure when declaring explicit interface implementations:

- **CS0071**: *An explicit interface implementation of an event must use event accessor syntax.*
- **CS0106**: *The modifier is not valid for this item.*
- **CS0531**: *Interface members cannot have a definition.*
- **CS0538**: *Member in explicit interface declaration is not an interface.*
- **CS0541**: *Explicit interface declaration can only be declared in a class, record, struct or interface.*

You can correct these errors using the following techniques:

- You must manually provide `add` and `remove` event accessors when explicitly implementing an interface event (**CS0071**). The compiler doesn't automatically generate these accessors for explicit interface implementations, so you must define them explicitly to specify how the event is stored and managed.
- Remove the `public` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations are implicitly public when accessed through the interface type, making the `public` keyword redundant and not allowed in this context.
- Remove the `abstract` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations provide the actual implementation and can't be marked as abstract because they can't be overridden in derived classes.
- Remove the method body from interface member declarations, or move the implementation to a class or struct that implements the interface (**CS0531**). Before C# 8.0, interface members can't contain implementations; starting with C# 8.0, you can provide default interface methods using specific syntax.
- Verify that the type specified in the explicit interface declaration is an actual interface type (**CS0538**). Only interface types can be used in explicit interface implementation syntax; attempting to use a class or other non-interface type violates the explicit implementation rules.
- Move explicit interface declarations into a class or struct that declares the interface in its base list (**CS0541**). Explicit interface implementations must appear within the body of a class or struct type and can't be declared at the namespace level or in other contexts.

For more information, see Interfaces, Explicit Interface Implementation, and How to implement interface events.

Return types and signatures

The following errors occur when the implementing method's signature doesn't match the interface member declaration:

- **CS0738**: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- **CS8854**: *Member does not implement interface member.*
- **CS9333**: *Parameter type must match implemented member.*
- **CS9334**: *Return type must match implemented member.*

You can correct these errors using the following techniques:

- Change the return type of the implementing method to exactly match the return type declared in the interface member (**CS0738**, **CS9334**). The signature of the implementation must match the interface declaration precisely because the method signature is part of the contract that determines which interface member is being implemented.
- Ensure that parameter types in the implementing method exactly match the parameter types declared in the interface member (**CS9333**). Each parameter must have the identical type in the same position as specified in the interface declaration, as parameter types are fundamental components of the method signature that the compiler uses to match implementations to interface members.
- Add an `init` accessor to the implementing property when the interface property declares an `init` setter (**CS8854**). The `init` keyword allows property initialization during object construction while preventing modification afterward, and the implementing property must provide this same initialization-only behavior to satisfy the interface contract.

For more information, see Interfaces, Properties, and Init-only setters.

Missing or incomplete implementations

The following errors occur when a class fails to fully implement an interface or implements members that don't match the interface contract:

- **CS0535**: *Member does not implement interface member.*
- **CS0550**: *Member adds an accessor not found in interface member.*
- **CS0551**: *Explicit interface implementation is missing an accessor.*

You can correct these errors using the following techniques:

- Provide an implementation for every member declared in the interface, or declare the type as `abstract` (**CS0535**). Each member must be implemented to satisfy the interface requirements.
- Remove any accessors from the implementing property that aren't declared in the interface property (**CS0550**). The implementing property can only include the accessors explicitly declared in the interface definition, ensuring that the implementation doesn't add functionality beyond what the interface contract specifies.
- Add all required accessors to the explicit interface implementation to match the interface declaration (**CS0551**). Each accessor declared in the interface must have a corresponding accessor in the implementation with matching signatures, as the implementation must fulfill the complete accessor contract defined by the interface.

For more information, see Interfaces and Properties.

Member matching and resolution

The following errors occur when attempting to implement interface members that don't exist in the interface or when the containing type doesn't declare the interface:

- **CS0539**: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- **CS0540**: *Containing type does not implement interface member.*

You can correct these errors using the following techniques:

- Verify that the member name and signature in the explicit interface implementation exactly match a member declared in the interface, or remove the incorrect implementation (**CS0539**). The member you're attempting to implement must actually exist in the interface definition with matching name, return type, and parameter types, as explicit interface implementation requires precise correspondence with the interface contract.
- Add the interface to the class's or struct's base list, or remove the explicit interface implementation (**CS0540**). A type can only explicitly implement members of interfaces that it declares in its inheritance list, so the implementing type must establish the interface relationship before it can provide explicit implementations.

For more information, see Interfaces and Explicit Interface Implementation.

Generic type constraints

The following errors occur when implementing generic interface methods with type parameter constraints:

- **CS0425**: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- **CS0460**: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*

You can correct these errors using the following techniques:

- Ensure the `where` clause in the implementing method is identical to the interface method declaration, or matches the semantic meaning of the constraints (**CS0425**). The type parameter constraints in the implementation must match those defined in the interface or base method.
- Remove explicit constraint declarations from override and explicit interface implementation methods (**CS0460**). The override method inherits its constraints automatically from the base or interface method, so redeclaring them is redundant and not permitted except for specific cases allowed in C# 8 and later.
- Apply the `default` constraint to resolve ambiguities with nullable reference types in override and explicit interface implementations when using C# 9 or later (**CS0460**). This exception to the constraint inheritance rule allows you to explicitly specify the default constraint to disambiguate nullable annotation contexts.
- Explicitly specify `where T : class` or `where T : struct` constraints on override and explicit interface implementation methods when using C# 8 or later to enable nullable reference type annotations (**CS0460**). These specific constraints are permitted to support nullable reference type analysis on type parameters that are constrained to reference or value types.

For more information, see Constraints on type parameters, Interfaces, and Nullable reference types.

Method visibility and modifiers

The following errors occur when implementing interface methods with incorrect accessibility or modifiers:

- **CS0736**: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- **CS0737**: *Member does not implement interface member. It cannot implement an interface member because it is not public.*

You can correct these errors using the following techniques:

- Remove the `static` modifier from the method declaration that implements the interface member (**CS0736**). Before C# 10, interface members are instance members, not static members.
- Add the `public` access modifier to the method that implements the interface member (**CS0737**). All interface members are implicitly `public` because interfaces define a contract for public behavior, so the implementing method must also have public accessibility to be accessible through the interface reference.

For more information, see Interfaces and Access Modifiers.

Accessor implementation and conflicts

The following errors occur when implementing interface properties or events with accessor methods that have visibility issues or naming conflicts:

- **CS0277**: *Member does not implement interface member because it is not public.*
- **CS0470**: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- **CS0686**: *Accessor cannot implement interface member. Use an explicit interface implementation.*

You can correct these errors using the following techniques:

- Remove any access modifiers from property accessors that restrict visibility to less visible than `public`, or add the `public` modifier if it's missing (**CS0277**). All interface members are implicitly `public`, so the implementing accessor must also have public accessibility to satisfy the interface contract and be accessible through the interface type.
- Replace methods with accessor-like names (such as `get_PropertyName`) with proper property syntax using explicit interface implementation (**CS0470**). The compiler generates accessor methods internally for properties, and attempting to manually create methods with these reserved names conflicts with the property implementation mechanism.
- Use explicit interface implementation syntax to resolve naming conflicts when the interface contains method names that match the auto-generated accessor methods (**CS0686**). The compiler automatically generates methods like `get_Property` and `set_Property` for properties, and `add_Event` and `remove_Event` for events, so if an interface declares methods with these exact names, explicit implementation is required to disambiguate between the interface method and the compiler-generated accessor.

For more information, see Interfaces, Properties, and Events.

Ambiguous and conflicting implementations

The following errors occur when the compiler can't determine which interface implementation to use:

- **CS0473**: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- **CS8705**: *Interface member 'member' does not have a most specific implementation. Neither is most specific.*

You can correct these errors using the following techniques:

- Eliminate the explicit interface implementation and instead use a single implicit public implementation for both interface methods (**CS0473**). When a generic method acquires the same signature as a non-generic method (such as when implementing `ITest<int>` where both `TestMethod(int)` and `TestMethod(T)` become identical), the common language infrastructure metadata system can't unambiguously determine which interface member binds to which implementation slot, so using implicit implementation allows the single method to satisfy both interface requirements.
- Provide an explicit implementation in the implementing class or struct that resolves the ambiguity between multiple default implementations (**CS8705**). This error typically occurs with diamond inheritance patterns where a class implements multiple interfaces that each provide default implementations for the same member. The compiler needs you to explicitly specify which implementation to use, or provide your own implementation.
- Restructure the interface hierarchy to avoid diamond inheritance conflicts where multiple interfaces provide default implementations for the same member (**CS8705**). By redesigning the interface relationships or consolidating the default implementations into a single interface, you can eliminate the ambiguity that prevents the compiler from determining the most specific implementation.

For more information, see Interfaces and Default Interface Methods.

Special implementation restrictions

The following error occurs when using special parameter types that aren't compatible with interface implementation:

- **CS0630**: *Member cannot implement interface member because it has an __arglist parameter.*

You can correct this error using the following techniques:

- Remove the `__arglist` parameter from the implementing method (**CS0630**). The `__arglist` keyword allows methods to accept variable numbers of arguments in an unmanaged way, but this feature is incompatible with interface implementation because interface contracts require predictable, type-safe signatures that can be verified at compile time.
- Replace the `__arglist` parameter with a `params` array parameter for variable-length argument lists (**CS0630**). Unlike `__arglist`, the `params` keyword provides a type-safe mechanism for accepting variable numbers of arguments that is fully compatible with interface implementation and maintains the compile-time type safety that interfaces require.

For more information, see Interfaces and params keyword.

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

---

Last updated on 11/14/2025

Resolve errors and warnings related to members that implement an interface

This article covers the following compiler errors:

- CS0071: *An explicit interface implementation of an event must use event accessor syntax.*
- CS0106: *The modifier is not valid for this item.*
- CS0277: *Member does not implement interface member because it is not public.*
- CS0425: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- CS0460: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*
- CS0470: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- CS0473: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- CS0531: *Interface members cannot have a definition.*
- CS0535: *Member does not implement interface member.*
- CS0538: *Member in explicit interface declaration is not an interface.*
- CS0539: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- CS0540: *Containing type does not implement interface member.*
- CS0541: *Explicit interface declaration can only be declared in a class, record, struct or interface.*
- CS0550: *Member adds an accessor not found in interface member.*
- CS0551: *Explicit interface implementation is missing an accessor.*
- CS0630: *Member cannot implement interface member because it has an __arglist parameter.*
- CS0686: *Accessor cannot implement interface member. Use an explicit interface implementation.*
- CS0736: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- CS0737: *Member does not implement interface member. It cannot implement an interface member because it is not public.*
- CS0738: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- CS8705: *Interface member does not have a most specific implementation. Neither member is most specific.*

- CS8854: *Member does not implement interface member.*
- CS9333: *Parameter type must match implemented member.*
- CS9334: *Return type must match implemented member.*

Interface declaration and syntax

The following errors relate to proper syntax and structure when declaring explicit interface implementations:

- **CS0071**: *An explicit interface implementation of an event must use event accessor syntax.*
- **CS0106**: *The modifier is not valid for this item.*
- **CS0531**: *Interface members cannot have a definition.*
- **CS0538**: *Member in explicit interface declaration is not an interface.*
- **CS0541**: *Explicit interface declaration can only be declared in a class, record, struct or interface.*

You can correct these errors using the following techniques:

- You must manually provide `add` and `remove` event accessors when explicitly implementing an interface event (**CS0071**). The compiler doesn't automatically generate these accessors for explicit interface implementations, so you must define them explicitly to specify how the event is stored and managed.
- Remove the `public` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations are implicitly public when accessed through the interface type, making the `public` keyword redundant and not allowed in this context.
- Remove the `abstract` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations provide the actual implementation and can't be marked as abstract because they can't be overridden in derived classes.
- Remove the method body from interface member declarations, or move the implementation to a class or struct that implements the interface (**CS0531**). Before C# 8.0, interface members can't contain implementations; starting with C# 8.0, you can provide default interface methods using specific syntax.
- Verify that the type specified in the explicit interface declaration is an actual interface type (**CS0538**). Only interface types can be used in explicit interface implementation syntax; attempting to use a class or other non-interface type violates the explicit implementation rules.
- Move explicit interface declarations into a class or struct that declares the interface in its base list (**CS0541**). Explicit interface implementations must appear within the body of a class or struct type and can't be declared at the namespace level or in other contexts.

For more information, see Interfaces, Explicit Interface Implementation, and How to implement interface events.

Return types and signatures

The following errors occur when the implementing method's signature doesn't match the interface member declaration:

- **CS0738**: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- **CS8854**: *Member does not implement interface member.*
- **CS9333**: *Parameter type must match implemented member.*
- **CS9334**: *Return type must match implemented member.*

You can correct these errors using the following techniques:

- Change the return type of the implementing method to exactly match the return type declared in the interface member (**CS0738**, **CS9334**). The signature of the implementation must match the interface declaration precisely because the method signature is part of the contract that determines which interface member is being implemented.
- Ensure that parameter types in the implementing method exactly match the parameter types declared in the interface member (**CS9333**). Each parameter must have the identical type in the same position as specified in the interface declaration, as parameter types are fundamental components of the method signature that the compiler uses to match implementations to interface members.
- Add an `init` accessor to the implementing property when the interface property declares an `init` setter (**CS8854**). The `init` keyword allows property initialization during object construction while preventing modification afterward, and the implementing property must provide this same initialization-only behavior to satisfy the interface contract.

For more information, see Interfaces, Properties, and Init-only setters.

Missing or incomplete implementations

The following errors occur when a class fails to fully implement an interface or implements members that don't match the interface contract:

- **CS0535**: *Member does not implement interface member.*
- **CS0550**: *Member adds an accessor not found in interface member.*
- **CS0551**: *Explicit interface implementation is missing an accessor.*

You can correct these errors using the following techniques:

- Provide an implementation for every member declared in the interface, or declare the type as `abstract` (**CS0535**). Each member must be implemented to satisfy the interface requirements.
- Remove any accessors from the implementing property that aren't declared in the interface property (**CS0550**). The implementing property can only include the accessors explicitly declared in the interface definition, ensuring that the implementation doesn't add functionality beyond what the interface contract specifies.
- Add all required accessors to the explicit interface implementation to match the interface declaration (**CS0551**). Each accessor declared in the interface must have a corresponding accessor in the implementation with matching signatures, as the implementation must fulfill the complete accessor contract defined by the interface.

For more information, see Interfaces and Properties.

Member matching and resolution

The following errors occur when attempting to implement interface members that don't exist in the interface or when the containing type doesn't declare the interface:

- **CS0539**: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- **CS0540**: *Containing type does not implement interface member.*

You can correct these errors using the following techniques:

- Verify that the member name and signature in the explicit interface implementation exactly match a member declared in the interface, or remove the incorrect implementation (**CS0539**). The member you're attempting to implement must actually exist in the interface definition with matching name, return type, and parameter types, as explicit interface implementation requires precise correspondence with the interface contract.
- Add the interface to the class's or struct's base list, or remove the explicit interface implementation (**CS0540**). A type can only explicitly implement members of interfaces that it declares in its inheritance list, so the implementing type must establish the interface relationship before it can provide explicit implementations.

For more information, see Interfaces and Explicit Interface Implementation.

Generic type constraints

The following errors occur when implementing generic interface methods with type parameter constraints:

- **CS0425**: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- **CS0460**: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*

You can correct these errors using the following techniques:

- Ensure the `where` clause in the implementing method is identical to the interface method declaration, or matches the semantic meaning of the constraints (**CS0425**). The type parameter constraints in the implementation must match those defined in the interface or base method.
- Remove explicit constraint declarations from override and explicit interface implementation methods (**CS0460**). The override method inherits its constraints automatically from the base or interface method, so redeclaring them is redundant and not permitted except for specific cases allowed in C# 8 and later.
- Apply the `default` constraint to resolve ambiguities with nullable reference types in override and explicit interface implementations when using C# 9 or later (**CS0460**). This exception to the constraint inheritance rule allows you to explicitly specify the default constraint to disambiguate nullable annotation contexts.
- Explicitly specify `where T : class` or `where T : struct` constraints on override and explicit interface implementation methods when using C# 8 or later to enable nullable reference type annotations (**CS0460**). These specific constraints are permitted to support nullable reference type analysis on type parameters that are constrained to reference or value types.

For more information, see Constraints on type parameters, Interfaces, and Nullable reference types.

Method visibility and modifiers

The following errors occur when implementing interface methods with incorrect accessibility or modifiers:

- **CS0736**: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- **CS0737**: *Member does not implement interface member. It cannot implement an interface member because it is not public.*

You can correct these errors using the following techniques:

- Remove the `static` modifier from the method declaration that implements the interface member (**CS0736**). Before C# 10, interface members are instance members, not static members.
- Add the `public` access modifier to the method that implements the interface member (**CS0737**). All interface members are implicitly `public` because interfaces define a contract for public behavior, so the implementing method must also have public accessibility to be accessible through the interface reference.

For more information, see Interfaces and Access Modifiers.

Accessor implementation and conflicts

The following errors occur when implementing interface properties or events with accessor methods that have visibility issues or naming conflicts:

- **CS0277**: *Member does not implement interface member because it is not public.*
- **CS0470**: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- **CS0686**: *Accessor cannot implement interface member. Use an explicit interface implementation.*

You can correct these errors using the following techniques:

- Remove any access modifiers from property accessors that restrict visibility to less visible than `public`, or add the `public` modifier if it's missing (**CS0277**). All interface members are implicitly `public`, so the implementing accessor must also have public accessibility to satisfy the interface contract and be accessible through the interface type.
- Replace methods with accessor-like names (such as `get_PropertyName`) with proper property syntax using explicit interface implementation (**CS0470**). The compiler generates accessor methods internally for properties, and attempting to manually create methods with these reserved names conflicts with the property implementation mechanism.
- Use explicit interface implementation syntax to resolve naming conflicts when the interface contains method names that match the auto-generated accessor methods (**CS0686**). The compiler automatically generates methods like `get_Property` and `set_Property` for properties, and `add_Event` and `remove_Event` for events, so if an interface declares methods with these exact names, explicit implementation is required to disambiguate between the interface method and the compiler-generated accessor.

For more information, see Interfaces, Properties, and Events.

Ambiguous and conflicting implementations

The following errors occur when the compiler can't determine which interface implementation to use:

- **CS0473**: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- **CS8705**: *Interface member 'member' does not have a most specific implementation. Neither is most specific.*

You can correct these errors using the following techniques:

- Eliminate the explicit interface implementation and instead use a single implicit public implementation for both interface methods (**CS0473**). When a generic method acquires the same signature as a non-generic method (such as when implementing `ITest<int>` where both `TestMethod(int)` and `TestMethod(T)` become identical), the common language infrastructure metadata system can't unambiguously determine which interface member binds to which implementation slot, so using implicit implementation allows the single method to satisfy both interface requirements.
- Provide an explicit implementation in the implementing class or struct that resolves the ambiguity between multiple default implementations (**CS8705**). This error typically occurs with diamond inheritance patterns where a class implements multiple interfaces that each provide default implementations for the same member. The compiler needs you to explicitly specify which implementation to use, or provide your own implementation.
- Restructure the interface hierarchy to avoid diamond inheritance conflicts where multiple interfaces provide default implementations for the same member (**CS8705**). By redesigning the interface relationships or consolidating the default implementations into a single interface, you can eliminate the ambiguity that prevents the compiler from determining the most specific implementation.

For more information, see Interfaces and Default Interface Methods.

Special implementation restrictions

The following error occurs when using special parameter types that aren't compatible with interface implementation:

- **CS0630**: *Member cannot implement interface member because it has an __arglist parameter.*

You can correct this error using the following techniques:

- Remove the `__arglist` parameter from the implementing method (**CS0630**). The `__arglist` keyword allows methods to accept variable numbers of arguments in an unmanaged way, but this feature is incompatible with interface implementation because interface contracts require predictable, type-safe signatures that can be verified at compile time.
- Replace the `__arglist` parameter with a `params` array parameter for variable-length argument lists (**CS0630**). Unlike `__arglist`, the `params` keyword provides a type-safe mechanism for accepting variable numbers of arguments that is fully compatible with interface implementation and maintains the compile-time type safety that interfaces require.

For more information, see Interfaces and params keyword.

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

---

Last updated on 11/14/2025

Compiler Error CS0542

Article • 06/18/2022

'user-defined type' : member names cannot be the same as their enclosing type

The members of a class or struct cannot have the same name as the class or struct, unless the member is a constructor.

The following sample generates CS0542:

```
C#
```

```
// CS0542.cs
class C
{
    public int C;
}
```

This error might be caused if you inadvertently put a return type on a constructor, which in effect makes it into an ordinary method. The following example generates CS0542 because `F` is a method, not a constructor, because it has a return type:

```
C#
```

```
// CS0542.cs
class F
{
   // Remove void from F() to resolve the problem.
   void F()   // CS0542, same name as the class
   {
   }
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0543

Article • 09/15/2021

'enumeration' : the enumerator value is too large to fit in its type

A value that was assigned to an element in an enumeration is outside the range of the data type.

The following sample generates CS0543:

C#

```
// CS0543.cs
namespace x
{
   enum I : byte
   {a = 255, b, c}   // CS0543
   public class clx
   {
      public static int Main()
      {
         return 0;
      }
   }
}
```

Compiler Error CS0544

Article • 09/15/2021

'property override': cannot override because 'non-property' is not a property

An attempt was made to override a nonproperty data type as a property, which is not allowed.

The following sample generates CS0544:

C#

```
// CS0544.cs
// compile with: /target:library
public class a
{
   public int i;
}

public class b : a
{
   public override int i {   // CS0544
   // try the following line instead
   // public new int i {
      get
      {
         return 0;
      }
   }
}
```

Compiler Error CS0545

Article • 09/15/2021

'function' : cannot override because 'property' does not have an overridable get accessor

A try was made to define an override for a property accessor when the base class has no such definition to override. You can resolve this error by:

- Adding a `set` accessor in the base class.
- Removing the `set` accessor from the derived class.
- Hiding the base class property by adding the new keyword to a property in a derived class.
- Making the base class property virtual.

For more information, see Using Properties.

Example

The following sample generates CS0545.

C#

```
// CS0545.cs
// compile with: /target:library
// CS0545
public class a
{
   public virtual int i
   {
      set {}

      // Uncomment the following line to resolve.
      // get { return 0; }
   }
}

public class b : a
{
   public override int i
   {
      get { return 0; }
      set {}   // OK
```

```
    }
}
```

Compiler Error CS0546

Article • 09/15/2021

'accessor' : cannot override because 'property' does not have an overridable set accessor

An attempt to override one of the accessor methods for a property failed because the accessor cannot be overridden. This error can occur if:

- the base class property is not declared as virtual.
- the base class property does not declare the get or set accessor you are trying to override.

If you do not want to override the base class property, you can use the new keyword before the property in derived class.

For more information, see Using Properties.

Example

The following sample generates CS0546 because the base class does not declare a set accessor for the property.

C#

```
// CS0546.cs
// compile with: /target:library
public class a
{
   public virtual int i
   {
      get
      {
         return 0;
      }
   }

   public virtual int i2
   {
      get
      {
         return 0;
      }

      set {}
   }
}
```

```
public class b : a
{
   public override int i
   {
      set {}   // CS0546 error no set
   }

   public override int i2
   {
      set {}   // OK
   }
}
```

See also

- Properties

Compiler Error CS0547

Article • 09/15/2021

'property' : property or indexer cannot have void type

void is invalid as a return value for a property.

For more information, see Properties.

The following sample generates CS0547:

C#

```
// CS0547.cs
public class a
{
   public void i   // CS0547
   // Try the following declaration instead:
   // public int i
   {
      get
      {
         return 0;
      }
   }
}

public class b : a
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0548

Article • 09/15/2021

'property' : property or indexer must have at least one accessor

A property must have at least one accessor (get or set) method.

For more information, see and Using Properties.

Example

The following sample generates CS0548.

C#

```
// CS0548.cs
// compile with: /target:library
public class b
{
   public int MyProp {}   // CS0548

   public int MyProp2   // OK
   {
      get
      {
         return 0;
      }
      set {}
   }
}
```

Compiler Error CS0549

Article • 09/15/2021

'function' is a new virtual member in sealed class 'class'

A sealedclass cannot be used as a base class. Therefore, it is useless to have a virtual method in sealed class.

The following sample generates CS0549:

C#

```
// CS0549.cs
// compile with: /target:library
sealed public class MyClass
{
   virtual public void TestMethod() {}   // CS0549
   public void TestMethod2() {}   // OK
}
```

Resolve errors and warnings related to members that implement an interface

This article covers the following compiler errors:

- CS0071: *An explicit interface implementation of an event must use event accessor syntax.*
- CS0106: *The modifier is not valid for this item.*
- CS0277: *Member does not implement interface member because it is not public.*
- CS0425: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- CS0460: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*
- CS0470: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- CS0473: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- CS0531: *Interface members cannot have a definition.*
- CS0535: *Member does not implement interface member.*
- CS0538: *Member in explicit interface declaration is not an interface.*
- CS0539: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- CS0540: *Containing type does not implement interface member.*
- CS0541: *Explicit interface declaration can only be declared in a class, record, struct or interface.*
- CS0550: *Member adds an accessor not found in interface member.*
- CS0551: *Explicit interface implementation is missing an accessor.*
- CS0630: *Member cannot implement interface member because it has an __arglist parameter.*
- CS0686: *Accessor cannot implement interface member. Use an explicit interface implementation.*
- CS0736: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- CS0737: *Member does not implement interface member. It cannot implement an interface member because it is not public.*
- CS0738: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- CS8705: *Interface member does not have a most specific implementation. Neither member is most specific.*

- CS8854: *Member does not implement interface member.*
- CS9333: *Parameter type must match implemented member.*
- CS9334: *Return type must match implemented member.*

Interface declaration and syntax

The following errors relate to proper syntax and structure when declaring explicit interface implementations:

- **CS0071**: *An explicit interface implementation of an event must use event accessor syntax.*
- **CS0106**: *The modifier is not valid for this item.*
- **CS0531**: *Interface members cannot have a definition.*
- **CS0538**: *Member in explicit interface declaration is not an interface.*
- **CS0541**: *Explicit interface declaration can only be declared in a class, record, struct or interface.*

You can correct these errors using the following techniques:

- You must manually provide `add` and `remove` event accessors when explicitly implementing an interface event (**CS0071**). The compiler doesn't automatically generate these accessors for explicit interface implementations, so you must define them explicitly to specify how the event is stored and managed.
- Remove the `public` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations are implicitly public when accessed through the interface type, making the `public` keyword redundant and not allowed in this context.
- Remove the `abstract` modifier from explicit interface implementations (**CS0106**). Explicit interface implementations provide the actual implementation and can't be marked as abstract because they can't be overridden in derived classes.
- Remove the method body from interface member declarations, or move the implementation to a class or struct that implements the interface (**CS0531**). Before C# 8.0, interface members can't contain implementations; starting with C# 8.0, you can provide default interface methods using specific syntax.
- Verify that the type specified in the explicit interface declaration is an actual interface type (**CS0538**). Only interface types can be used in explicit interface implementation syntax; attempting to use a class or other non-interface type violates the explicit implementation rules.
- Move explicit interface declarations into a class or struct that declares the interface in its base list (**CS0541**). Explicit interface implementations must appear within the body of a class or struct type and can't be declared at the namespace level or in other contexts.

For more information, see Interfaces, Explicit Interface Implementation, and How to implement interface events.

Return types and signatures

The following errors occur when the implementing method's signature doesn't match the interface member declaration:

- **CS0738**: *Member does not implement interface member. It cannot because it does not have the matching return type.*
- **CS8854**: *Member does not implement interface member.*
- **CS9333**: *Parameter type must match implemented member.*
- **CS9334**: *Return type must match implemented member.*

You can correct these errors using the following techniques:

- Change the return type of the implementing method to exactly match the return type declared in the interface member (**CS0738**, **CS9334**). The signature of the implementation must match the interface declaration precisely because the method signature is part of the contract that determines which interface member is being implemented.
- Ensure that parameter types in the implementing method exactly match the parameter types declared in the interface member (**CS9333**). Each parameter must have the identical type in the same position as specified in the interface declaration, as parameter types are fundamental components of the method signature that the compiler uses to match implementations to interface members.
- Add an `init` accessor to the implementing property when the interface property declares an `init` setter (**CS8854**). The `init` keyword allows property initialization during object construction while preventing modification afterward, and the implementing property must provide this same initialization-only behavior to satisfy the interface contract.

For more information, see Interfaces, Properties, and Init-only setters.

Missing or incomplete implementations

The following errors occur when a class fails to fully implement an interface or implements members that don't match the interface contract:

- **CS0535**: *Member does not implement interface member.*
- **CS0550**: *Member adds an accessor not found in interface member.*
- **CS0551**: *Explicit interface implementation is missing an accessor.*

You can correct these errors using the following techniques:

- Provide an implementation for every member declared in the interface, or declare the type as `abstract` (**CS0535**). Each member must be implemented to satisfy the interface requirements.
- Remove any accessors from the implementing property that aren't declared in the interface property (**CS0550**). The implementing property can only include the accessors explicitly declared in the interface definition, ensuring that the implementation doesn't add functionality beyond what the interface contract specifies.
- Add all required accessors to the explicit interface implementation to match the interface declaration (**CS0551**). Each accessor declared in the interface must have a corresponding accessor in the implementation with matching signatures, as the implementation must fulfill the complete accessor contract defined by the interface.

For more information, see Interfaces and Properties.

Member matching and resolution

The following errors occur when attempting to implement interface members that don't exist in the interface or when the containing type doesn't declare the interface:

- **CS0539**: *Member in explicit interface declaration is not found among members of the interface that can be implemented.*
- **CS0540**: *Containing type does not implement interface member.*

You can correct these errors using the following techniques:

- Verify that the member name and signature in the explicit interface implementation exactly match a member declared in the interface, or remove the incorrect implementation (**CS0539**). The member you're attempting to implement must actually exist in the interface definition with matching name, return type, and parameter types, as explicit interface implementation requires precise correspondence with the interface contract.
- Add the interface to the class's or struct's base list, or remove the explicit interface implementation (**CS0540**). A type can only explicitly implement members of interfaces that it declares in its inheritance list, so the implementing type must establish the interface relationship before it can provide explicit implementations.

For more information, see Interfaces and Explicit Interface Implementation.

Generic type constraints

The following errors occur when implementing generic interface methods with type parameter constraints:

- **CS0425**: *The constraints for type parameter of method must match the constraints for type parameter of interface method. Consider using an explicit interface implementation instead.*
- **CS0460**: *Constraints for override and explicit interface implementation methods are inherited from the base method, so they cannot be specified directly, except for either a 'class', or a 'struct' constraint.*

You can correct these errors using the following techniques:

- Ensure the `where` clause in the implementing method is identical to the interface method declaration, or matches the semantic meaning of the constraints (**CS0425**). The type parameter constraints in the implementation must match those defined in the interface or base method.
- Remove explicit constraint declarations from override and explicit interface implementation methods (**CS0460**). The override method inherits its constraints automatically from the base or interface method, so redeclaring them is redundant and not permitted except for specific cases allowed in C# 8 and later.
- Apply the `default` constraint to resolve ambiguities with nullable reference types in override and explicit interface implementations when using C# 9 or later (**CS0460**). This exception to the constraint inheritance rule allows you to explicitly specify the default constraint to disambiguate nullable annotation contexts.
- Explicitly specify `where T : class` or `where T : struct` constraints on override and explicit interface implementation methods when using C# 8 or later to enable nullable reference type annotations (**CS0460**). These specific constraints are permitted to support nullable reference type analysis on type parameters that are constrained to reference or value types.

For more information, see Constraints on type parameters, Interfaces, and Nullable reference types.

Method visibility and modifiers

The following errors occur when implementing interface methods with incorrect accessibility or modifiers:

- **CS0736**: *Member does not implement instance interface member. It cannot implement the interface member because it is static.*
- **CS0737**: *Member does not implement interface member. It cannot implement an interface member because it is not public.*

You can correct these errors using the following techniques:

- Remove the `static` modifier from the method declaration that implements the interface member (**CS0736**). Before C# 10, interface members are instance members, not static members.
- Add the `public` access modifier to the method that implements the interface member (**CS0737**). All interface members are implicitly `public` because interfaces define a contract for public behavior, so the implementing method must also have public accessibility to be accessible through the interface reference.

For more information, see Interfaces and Access Modifiers.

Accessor implementation and conflicts

The following errors occur when implementing interface properties or events with accessor methods that have visibility issues or naming conflicts:

- **CS0277**: *Member does not implement interface member because it is not public.*
- **CS0470**: *Method cannot implement interface accessor for type. Use an explicit interface implementation.*
- **CS0686**: *Accessor cannot implement interface member. Use an explicit interface implementation.*

You can correct these errors using the following techniques:

- Remove any access modifiers from property accessors that restrict visibility to less visible than `public`, or add the `public` modifier if it's missing (**CS0277**). All interface members are implicitly `public`, so the implementing accessor must also have public accessibility to satisfy the interface contract and be accessible through the interface type.
- Replace methods with accessor-like names (such as `get_PropertyName`) with proper property syntax using explicit interface implementation (**CS0470**). The compiler generates accessor methods internally for properties, and attempting to manually create methods with these reserved names conflicts with the property implementation mechanism.
- Use explicit interface implementation syntax to resolve naming conflicts when the interface contains method names that match the auto-generated accessor methods (**CS0686**). The compiler automatically generates methods like `get_Property` and `set_Property` for properties, and `add_Event` and `remove_Event` for events, so if an interface declares methods with these exact names, explicit implementation is required to disambiguate between the interface method and the compiler-generated accessor.

For more information, see Interfaces, Properties, and Events.

Ambiguous and conflicting implementations

The following errors occur when the compiler can't determine which interface implementation to use:

- **CS0473**: *Explicit interface implementation 'method name' matches more than one interface member. Which interface member is actually chosen is implementation-dependent. Consider using a non-explicit implementation instead.*
- **CS8705**: *Interface member 'member' does not have a most specific implementation. Neither is most specific.*

You can correct these errors using the following techniques:

- Eliminate the explicit interface implementation and instead use a single implicit public implementation for both interface methods (**CS0473**). When a generic method acquires the same signature as a non-generic method (such as when implementing `ITest<int>` where both `TestMethod(int)` and `TestMethod(T)` become identical), the common language infrastructure metadata system can't unambiguously determine which interface member binds to which implementation slot, so using implicit implementation allows the single method to satisfy both interface requirements.
- Provide an explicit implementation in the implementing class or struct that resolves the ambiguity between multiple default implementations (**CS8705**). This error typically occurs with diamond inheritance patterns where a class implements multiple interfaces that each provide default implementations for the same member. The compiler needs you to explicitly specify which implementation to use, or provide your own implementation.
- Restructure the interface hierarchy to avoid diamond inheritance conflicts where multiple interfaces provide default implementations for the same member (**CS8705**). By redesigning the interface relationships or consolidating the default implementations into a single interface, you can eliminate the ambiguity that prevents the compiler from determining the most specific implementation.

For more information, see Interfaces and Default Interface Methods.

Special implementation restrictions

The following error occurs when using special parameter types that aren't compatible with interface implementation:

- **CS0630**: *Member cannot implement interface member because it has an __arglist parameter.*

You can correct this error using the following techniques:

- Remove the `__arglist` parameter from the implementing method (**CS0630**). The `__arglist` keyword allows methods to accept variable numbers of arguments in an unmanaged way, but this feature is incompatible with interface implementation because interface contracts require predictable, type-safe signatures that can be verified at compile time.
- Replace the `__arglist` parameter with a `params` array parameter for variable-length argument lists (**CS0630**). Unlike `__arglist`, the `params` keyword provides a type-safe mechanism for accepting variable numbers of arguments that is fully compatible with interface implementation and maintains the compile-time type safety that interfaces require.

For more information, see Interfaces and params keyword.

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

---

Last updated on 11/14/2025

Compiler Error CS0569

Article • 09/15/2021

'method2' : cannot override 'method1' because it is not supported by the language

This error occurs when you derive from a base class that was written in another language and when the compiler does not support the method that you are attempting to override.

Compiler Error CS0570

07/22/2025

Property, indexer, or event 'name' is not supported by the language; try directly calling accessor method 'name!'

This error occurs when using imported metadata that was generated by another compiler. Your code attempted to use a class member that the compiler cannot process.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

Example 1

The following C++ program uses an attribute, RequiredAttributeAttribute, which may not be consumed by other languages.

C++

```
// CPP0570.cpp
// compile with: /clr /LD

using namespace System;
using namespace System::Runtime::CompilerServices;

namespace CS0570_Server {
   [RequiredAttributeAttribute(Int32::typeid)]
   public ref struct Scenario1 {
      int intVar;
   };

   public ref struct CS0570Class {
      Scenario1 ^ sc1_field;

      property virtual Scenario1 ^ sc1_prop {
         Scenario1 ^ get() { return sc1_field; }
      }

      Scenario1 ^ sc1_method() { return sc1_field; }
   };
};
```

Example 2

The following sample generates CS0570.

C#

```
// CS0570.cs
// compile with: /reference:CPP0570.dll
using System;
using CS0570_Server;

public class C {
   public static int Main() {
      CS0570Class r = new CS0570Class();
      r.sc1_field = null;   // CS0570
      object o = r.sc1_prop;   // CS0570
      r.sc1_method();   // CS0570
   }
}
```

Compiler Error CS0571

Article • 09/15/2021

'function' : cannot explicitly call operator or accessor

Certain operators have internal names. For example, **op_Increment** is the internal name of the ++ operator. You should not use or explicitly call such method names.

The following sample generates CS0571:

C#

```
// CS0571.cs
public class MyClass
{
   public static MyClass operator ++ (MyClass c)
   {
      return null;
   }

   public static int prop
   {
      get
      {
         return 1;
      }
      set
      {
      }
   }

   public static void Main()
   {
      op_Increment(null);   // CS0571
      // use the increment operator as follows
      // MyClass x = new MyClass();
      // x++;

      set_prop(1);      // CS0571
      // try the following line instead
      // prop = 1;
   }
}
```

Compiler Error CS0572

Article • 09/15/2021

'type' : cannot reference a type through an expression; try 'path_to_type' instead

An attempt was made to access a member of a class through an identifier, which is not permitted in this situation.

The following sample generates CS0572:

C#

```
// CS0572.cs
using System;
class C
{
   public class Inner
   {
      public static int v = 9;
   }
}

class D : C
{
   public static void Main()
   {
      C cValue = new C();
      Console.WriteLine(cValue.Inner.v);   // CS0572
      // try the following line instead
      // Console.WriteLine(C.Inner.v);
   }
}
```

Compiler Error CS0573

Article • 12/04/2024

'field declaration' : cannot have instance field initializers in structs

You can't initialize an instance field of a struct. Fields of value types will be initialized to their default values, and reference-type fields will be initialized to `null`.

> **Note**
>
> You can initialize a struct's instance field or property at its declaration. For more information, see the Struct initialization and default values section of the Structure types article.

Example

The following sample generates CS0573:

C#

```
// CS0573.cs
namespace x
{
    public class clx
    {
        public static void Main()
        {
        }
    }

    public struct cly
    {
        clx a = new clx();   // CS0573
        // clx a;            // OK
        int i = 7;           // CS0573
        // int i;            // OK
    }
}
```

Compiler Error CS0574

Article • 09/16/2021

Name of destructor must match name of class

The name of a finalizer must be the class name preceded by a tilde (~).

The following sample generates CS0574:

C#

```
// CS0574.cs
namespace x
{
   public class iii
   {
      ~iiii()   // CS0574
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0575

Article • 09/15/2021

Only class types can contain destructors

A struct cannot contain a finalizer.

The following sample generates CS0575:

C#

```
// CS0575.cs
namespace x
{
   public struct iii
   {
      ~iii()   // CS0575
      {
      }

      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0579

Article • 09/15/2021

Duplicate 'attribute' attribute

It is not possible to specify the same attribute more than once unless the attribute specifies **AllowMultiple=true** in its AttributeUsage.

Example

The following example generates CS0579.

C#

```
// CS0579.cs
using System;
public class MyAttribute : Attribute
{
}

[AttributeUsage(AttributeTargets.All,AllowMultiple=true)]
public class MyAttribute2 : Attribute
{
}

public class z
{
    [MyAttribute, MyAttribute]     // CS0579
    public void zz()
    {
    }

    [MyAttribute2, MyAttribute2]   // OK
    public void zzz()
    {
    }

    public static void Main()
    {
    }
}
```

Compiler Error CS0583

Article • 09/15/2021

Internal Compiler Error. An internal error has occurred in the compiler. To work around this problem, try simplifying or changing the program near the locations listed below. Locations at the top of the list are closer to the point at which the internal error occurred.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0584

Article • 09/15/2021

Internal Compiler Error: stage 'stage' symbol 'symbol'

Try to determine whether the compiler is failing because of its inability to parse unexpected syntax. If that is not the case, see Visual Studio feedback options.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0585

Article • 09/15/2021

Internal Compiler Error: stage 'stage'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. If that is not the case, see Visual Studio feedback options.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0586

Article • 09/15/2021

Internal Compiler Error: stage 'stage'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. If that is not the case, see Visual Studio feedback options.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0587

Article • 09/15/2021

Internal Compiler Error: stage 'stage'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. If this is not the case, see Visual Studio feedback options.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0588

Article • 09/15/2021

Internal Compiler Error: stage 'LEX'

Try to determine whether the compiler is failing because of its inability to parse unexpected syntax. If this is not the case, see Visual Studio feedback options.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0589

Article • 09/15/2021

Internal Compiler Error: stage 'PARSE'

Try to determine if the compiler is failing due to its inability to parse unexpected syntax. If that is not the case, see Visual Studio feedback options.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS0592

Article • 03/15/2023

Attribute 'attribute' is not valid on this declaration type. It is valid on 'type' declarations only.

When you define an attribute, you define what constructs it can be applied to by specifying an `AttributeTargets` value. In the following example, the `MyAttribute` attribute can be applied to interfaces only, because the `AttributeUsage` attribute specifies `AttributeTargets.Interface`. The error is generated because the attribute is applied to a class (class `A`).

Example

The following sample generates CS0592:

C#

```
// CS0592.cs
using System;

[AttributeUsage(AttributeTargets.Interface)]
public class MyAttribute : Attribute
{
}

[MyAttribute]
// Generates CS0592 because MyAttribute is not valid for a class.
public class A
{
    public static void Main()
    {
    }
}
```

See also

- Attributes

Compiler Error CS0594

Article • 09/15/2021

Floating-point constant is outside the range of type 'type'

A value was assigned to a floating-point variable that is too large for the variables of this data type. See Integral Types Table for information about the range of values allowed in data types.

The following sample generates CS0594:

C#

```
// CS0594.cs
namespace MyNamespace
{
   public class MyClass
   {
      public static void Main()
      {
         float f = 6.77777777777E400;   // CS0594, value too large
      }
   }
}
```

Compiler Error CS0596

Article • 09/15/2021

The Guid attribute must be specified with the ComImport attribute

The **Guid** attribute must be present when using the **ComImport** attribute.

The following sample generates CS0596:

C#

```
// CS0596.cs
using System.Runtime.InteropServices;

namespace x
{
   [ComImport]   // CS0596
   // try the following line to resolve this CS0596
   // [ComImport, Guid("00000000-0000-0000-0000-000000000001")]
   public class a
   {
   }

   public class b
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS0601

Article • 09/15/2021

The DllImport attribute must be specified on a method marked 'static' and 'extern'

The `DllImport` attribute was used on a method that did not have the correct access keywords.

The following sample generates CS0601:

C#

```
// CS0601.cs
using System.Runtime.InteropServices;
using System.Text;

public class C
{
   [DllImport("KERNEL32.DLL")]
   extern int GetCurDirectory(int bufSize, StringBuilder buf);   // CS0601
   // Try the following line instead:
   // static extern int GetCurDirectory(int bufSize, StringBuilder buf);
}

public class MainClass
{
   public static void Main ()
   {
   }
}
```

Compiler Error CS0610

Article • 09/15/2021

Field or property cannot be of type 'type'

There are some types that cannot be used as fields or properties. These types include **System.ArgIterator** and **System.TypedReference**.

The following sample generates CS0610 as a result of using **System.TypedReference** as a field:

C#

```
// CS0610.cs
public class MainClass
{
   System.TypedReference i;   // CS0610
   public static void Main ()
   {
   }

   public static void Test(System.TypedReference i)   // OK
   {
   }
}
```

Compiler Error CS0616

Article • 09/15/2021

'class' is not an attribute class

An attempt was made to use a non-attribute class in an attribute block. All the attribute types need to be inherited from System.Attribute.

Example 1

The following sample generates CS0616.

C#

```
// CS0616.cs
// compile with: /target:library
[CMyClass(i = 5)]   // CS0616
public class CMyClass {}
```

Example 2

The following sample shows how you might define an attribute:

C#

```
// CreateAttrib.cs
// compile with: /target:library
using System;

[AttributeUsage(AttributeTargets.Class|AttributeTargets.Interface)]
public class MyAttr : Attribute
{
   public int Name = 0;
   public int Count = 0;

   public MyAttr (int iCount, int sName)
   {
      Count = iCount;
      Name = sName;
   }
}

[MyAttr(5, 50)]
class Class1 {}
```

```
[MyAttr(6, 60)]
interface Interface1 {}
```

Compiler Error CS0619

Article • 09/15/2021

'member' is obsolete: 'text'

A class member was marked with the Obsolete attribute, such that an error will be issued when the class member is referenced.

Example

The following sample generates CS0619:

C#

```
using System;

public class C
{
    [Obsolete("Use NewMethod instead", true)] // generates an error on use
    public static void OldMethod()
    {
    }

    // this is the method you should be using
    public static void NewMethod()
    {
    }
}

class MyClass
{
   public static void Main()
   {
      C.OldMethod();   // CS0619
   }
}
```

Compiler Error CS0620

Article • 09/15/2021

Indexers cannot have void type

The return type of an indexer cannot be `void`. An indexer must return a value.

The following sample generates CS0620:

C#

```
// CS0620.cs
class MyClass
{
   public static void Main()
   {
      MyClass test = new MyClass();
      System.Console.WriteLine(test[2]);
   }

   void this [int intI]   // CS0620, return type cannot be void
   {
      get
      {
         // will need to return some value
      }
   }
}
```

See also

- void

Compiler Error CS0621

Article • 09/15/2021

'member' : virtual or abstract members cannot be private

Private **virtual** or **abstract** members are not allowed.

The following sample generates CS0621:

C#

```
// CS0621.cs
abstract class MyClass
{
   private virtual void DoNothing1()   // CS0621
   {
   }

   private abstract void DoNothing2();   // CS0621

   public static void Main()
   {
   }
}
```

Compiler Error CS0622

Article • 09/15/2021

Can only use array initializer expressions to assign to array types. Try using a new expression instead.

Syntax that is appropriate to initialize an array was used in the declaration of a non-array.

Example

The following sample generates CS0622:

C#

```
// CS0622.cs
using System;

public class Test
{
    public static void Main ()
    {
        Test t = { new Test() };   // CS0622
        // Try the following instead:
        // Test[] t = { new Test() };
    }
}
```

Compiler Error CS0635

Article • 09/15/2021

'attribute' : System.Interop.UnmanagedType.CustomMarshaller requires named arguments ComType and Marshal

The **ComType** and **Marshal** arguments must be specified when the marshal format is **System.Interop.UnmanagedType.CustomMarshaller**.

Compiler Error CS0644

Article • 09/15/2021

'class1' cannot derive from special class 'class2'

Classes cannot explicitly inherit from any of the following base classes:

- **System.Enum**
- **System.ValueType**
- **System.Delegate**
- **System.Array**

These are used as implicit base classes by the compiler. For example, **System.ValueType** is the implicit base class of structs.

The following sample generates CS0644:

C#

```
// CS0644.cs
class MyClass : System.ValueType   // CS0644
{
}
```

Compiler Error CS0645

Article • 09/15/2021

Identifier too long

A class name or other identifier can be no longer than 512 characters.

Compiler Error CS0647

Article • 09/15/2021

Error emitting 'attribute' attribute -- 'reason'

The following sample generates CS0647:

C#

```
// CS0647.cs
using System.Runtime.InteropServices;

[Guid("z")]   // CS0647, incorrect uuid format.
// try the following line instead
// [Guid("00000000-0000-0000-0000-000000000001")]
public class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0648

Article • 09/15/2021

'type' is a type not supported by the language

Metadata that was generated from another language, possibly C++, contained a type that was not marked as managed. Information about this type is included in the metadata, but the type is not available to programs written in C#.

Compiler Error CS0656

07/22/2025

Missing compiler required member 'object.member'

One of the following problems exists:

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

- Your installation of the common language runtime is corrupt.
- You have a reference to an assembly that defines a type that is also found in the common language runtime. However, your assembly's type is not defined the way the C# compiler expects.

Check your references to ensure that you are using the correct version of the common language runtime.

Compiler Error CS0662

Article • 09/15/2021

'method' cannot specify only Out attribute on a ref parameter. Use both In and Out attributes, or neither.

An interface method has a parameter that uses ref with just the Out attribute. A `ref` parameter that uses the **Out** attribute must also use the In attribute.

The following sample generates CS0662:

C#

```
// CS0662.cs
using System.Runtime.InteropServices;

interface I
{
   void method([Out] ref int i);   // CS0662
   // try one of the following lines instead
   // void method(ref int i);
   // void method([Out, In]ref int i);
}

class test
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0664

Article • 09/15/2021

Literal of type double cannot be implicitly converted to type 'type'; use an 'suffix' suffix to create a literal of this type

An assignment could not be completed; use a suffix to correct the instruction. The documentation for each type identifies the corresponding suffix for the type. For more information on conversions, see Casting and Type Conversions.

The following sample generates CS0664:

C#

```
// CS0664.cs
class Example
{
    static void Main()
    {
        decimal d1 = 1.0;   // CS0664, because 1.0 is interpreted
                            // as a double.

        // Try the following line instead.
        decimal d2 = 1.0M;  // The M tells the compiler that 1.0 is a
                            // decimal.
        Console.WriteLine(d2);
    }
}
```

See also

- Type Conversion Tables

Compiler Error CS0666

Article • 09/15/2021

'member' : new protected member declared in struct

A struct cannot be abstract and is always implicitly sealed. Because structs do not support inheritance, the concept of a protected member in a struct makes no sense. For more information, see Inheritance.

Example

The following sample generates CS0666:

C#

```
// CS0666.cs
class M
{
    static void Main()
    {
    }
}

struct S
{
    protected int x;   // CS0666
}
```

Compiler Error CS0667

Article • 09/15/2021

The feature 'invalid feature' is deprecated. Please use 'valid feature' instead'.

The feature you are attempting to use is now deprecated. Update your code to use the valid feature instead.

Compiler Error CS0668

Article • 09/15/2021

Two indexers have different names; the IndexerName attribute must be used with the same name on every indexer within a type

The values passed to the **IndexerName** attribute must be the same for all indexers in a type. For more information on the **IndexerName** attribute, see IndexerNameAttribute Class.

The following sample generates CS0668:

C#

```
// CS0668.cs
using System;
using System.Runtime.CompilerServices;

class IndexerClass
{
   [IndexerName("IName1")]
   public int this [int index]   // indexer declaration
   {
      get
      {
         return index;
      }
      set
      {
      }
   }

   [IndexerName("IName2")]
   public int this [string s]    // CS0668, change IName2 to IName1
   {
      get
      {
         return int.Parse(s);
      }
      set
      {
      }
   }

   void Main()
   {
   }
}
```

Compiler Error CS0669

Article • 09/15/2021

A class with the ComImport attribute cannot have a user-defined constructor.

The COM interop layer in the common language runtime supplies the constructor for ComImport classes. Consequently, a COM object can be used as a managed object in the runtime.

The following sample generates CS0669:

C#

```
// CS0669.cs
using System.Runtime.InteropServices;
[ComImport, Guid("00000000-0000-0000-0000-000000000001")]
class TestClass
{
   TestClass()   // CS0669, delete constructor to resolve
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0670

Article • 09/15/2021

Field cannot have void type

A field was declared to be of type void.

The following sample generates CS0670:

C#

```
// CS0670.cs
class C
{
   void f;   // CS0670
   // try the following line instead
   // public int f;

   public static void Main()
   {
      C myc = new C();
      myc.f = 0;
   }
}
```

Compiler Error CS0673

Article • 09/15/2021

System.Void cannot be used from C# -- use typeof(void) to get the void type object.

System.Void cannot be used in C#.

The following sample generates CS0673:

C#

```
// CS0673.cs
class MyClass
{
   public static void Main()
   {
      System.Type t = typeof(System.Void);   // CS0673
      // try the following line instead
      // System.Type t = typeof(void);
   }
}
```

Compiler Error CS0677

08/30/2025

'variable': a volatile field cannot be of the type 'type'

Fields declared with the `volatile` keyword must be one of the following types:

- Any reference type.
- Any pointer type (in an `unsafe` context).
- The types `sbyte`, `byte`, `short`, `ushort`, `int`, `uint`, `char`, `float`, `bool`.
- Enum types based on any of the listed types.

The following sample generates CS0677:

C#

```
// CS0677.cs
class TestClass
{
   private volatile long i;   // CS0677

   public static void Main()
   {
   }
}
```

Potential workarounds

In some scenarios, you might be able to use `nint` (native-sized integer) instead of `long` as a workaround for CS0677. The `nint` type is guaranteed to support atomic access and can be used with the `volatile` keyword:

C#

```
class TestClass
{
   private volatile nint i;   // This compiles successfully

   public static void Main()
   {
   }
}
```

The `nint` type is a native-sized integer that's 32-bit on 32-bit platforms and 64-bit on 64-bit platforms. On 64-bit platforms, `nint` has the same size and range as `long`, but it's designed to guarantee atomic access. This workaround is most appropriate when:

- Your code targets 64-bit platforms where `nint` provides the same range as `long`.
- You need atomic access to a large integer value in a multithreaded context.
- Platform-specific integer size behavior is acceptable for your use case.

For more information about native-sized integers, see Integral numeric types.

As general guidance on thread-safe programming, consider using Interlocked operations or the lock statement instead of `volatile` fields.

Compiler Error CS0678

Article • 09/15/2021

'variable': a field can not be both volatile and readonly

Use of the volatile and readonly keywords is mutually exclusive.

The following sample generates CS0678:

C#

```
// CS0678.cs
using System;

class TestClass
{
   private readonly volatile int i;   // CS0678
   // try the following line instead
   // private volatile int i;

   public static void Main()
   {
   }
}
```

Compiler Error CS0681

Article • 09/15/2021

The modifier 'abstract' is not valid on fields. Try using a property instead

You cannot make a field abstract. You can, however, have an abstract property that accesses the field.

Example 1

The following sample generates CS0681:

C#

```
// CS0681.cs
// compile with: /target:library
abstract class C
{
    abstract int num;  // CS0681
}
```

Example 2

Try the following code instead:

C#

```
// CS0681b.cs
// compile with: /target:library
abstract class C
{
    public abstract int num
    {
       get;
       set;
    }
}
```

Compiler Error CS0682

Article • 09/15/2021

'type1' cannot implement 'type2' because it is not supported by the language

This error occurs when you try to implement an interface written in another language and the compiler does not support the interface.

Compiler Error CS0683

Article • 09/15/2021

'explicitmethod' explicit method implementation cannot implement 'method' because it is an accessor

The following sample generates CS0683:

C#

```
// CS0683.cs
interface IExample
{
   int Test { get; }
}

class CExample : IExample
{
   int IExample.get_Test() { return 0; } // CS0683
   int IExample.Test { get{ return 0; } } // correct
}
```

Compiler Error CS0689

Article • 09/15/2021

Cannot derive from 'identifier' because it is a type parameter

Base classes or interfaces for generic classes cannot be specified by a type parameter. Derive from a specific class or interface, or a specific generic class instead, or include the unknown type as a member.

The following sample generates CS0689:

C#

```
// CS0689.cs
class A<T> : T   // CS0689
{
}
```

Compiler Error CS0690

Article • 09/15/2021

Input file 'file' contains invalid metadata.

You are able to open the metadata file, but due to some localization problem, the metadata was corrupted. This is similar to error CS0009, except that you are able to open the metadata file.

A PE (Process Executable) is an executable file.

Compiler Error CS0692

Article • 09/15/2021

Duplicate type parameter 'identifier'

The same name may not be used more than once in a type parameter list. Rename or remove the duplicate type parameter(s).

Example

The following sample generates CS0692:

C#

```
// CS0692.cs
// compile with: /target:library
class C <T, A, T>   // CS0692
{
}

class D <T, T>   // CS0692
{
}
```

Resolve errors and warnings related to generic type parameters and generic type arguments

This article covers the following compiler errors:

- CS0080: *Constraints are not allowed on non-generic declarations.*
- CS0081: *Type parameter declaration must be an identifier not a type.*
- CS0224: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- CS0304: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- CS0305: *Using the generic type 'generic type' requires 'number' type arguments.*
- CS0306: *The type 'type' may not be used as a type argument.*
- CS0307: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- CS0308: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*
- CS0310: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- CS0311: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- CS0312: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- CS0313: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- CS0314: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- CS0315: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*
- CS0403: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- CS0412: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- CS0413: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*

- CS0417: *'identifier': cannot provide arguments when creating an instance of a variable type.*
- CS0694: *Type parameter 'identifier' has the same name as the containing type, or method.*
- CS0695: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- CS0698: *A generic type cannot derive from 'type' because it is an attribute class.*
- CS9338: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

Type parameter declaration and naming

The following errors relate to how type parameters are declared and named in generic types and methods:

- **CS0080**: *Constraints are not allowed on non-generic declarations.*
- **CS0081**: *Type parameter declaration must be an identifier not a type.*
- **CS0412**: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- **CS0694**: *Type parameter 'identifier' has the same name as the containing type, or method.*

To correct these errors, ensure that type parameters are declared with valid identifiers, constraint clauses are only applied to generic declarations, and type parameter names don't conflict with other identifiers in scope:

- Remove the constraint clause from non-generic declarations (**CS0080**). The `where` clause can only be used on generic types and methods that declare type parameters, because constraints define requirements that type arguments must satisfy. If you need to apply constraints, first add type parameters to your type or method declaration. For example, change `public class MyClass where MyClass : System.IDisposable` to `public class MyClass<T> where T : System.IDisposable`.
- Replace actual type names with identifiers in type parameter declarations (**CS0081**). Type parameters must be declared using identifiers (like `T`, `TKey`, or `TValue`) rather than concrete types (like `int` or `string`), because the purpose of a type parameter is to serve as a placeholder that is substituted with actual types when the generic type or method is used. For example, change `public void F<int>()` to `public void F<T>()`.
- Rename type parameters, local variables, or parameters to avoid naming conflicts (**CS0412**, **CS0694**). Type parameter names can't shadow identifiers in the same scope. They can't match the name of the containing type or method. Such conflicts create ambiguity about which identifier is being referenced. For example, if you have a method `public void F<T>()`, you can't declare a local variable `double T` inside that method, and you can't name a type parameter the same as its containing type (`class C<C>`).

For more information, see Generic Type Parameters and Generics.

Type argument count and usage

The following errors relate to providing the correct number and type of type arguments to generic types and methods:

- **CS0224**: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- **CS0305**: *Using the generic type 'generic type' requires 'number' type arguments.*
- **CS0306**: *The type 'type' may not be used as a type argument.*
- **CS0307**: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- **CS0308**: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*

To correct these errors, ensure that you provide the exact number of type arguments required by the generic declaration. Use only valid types as type arguments. Don't apply type arguments to non-generic constructs:

- Remove generic type parameters or containing generic type declarations from methods that use `__arglist` (**CS0224**). The `__arglist` keyword is incompatible with generics because the runtime mechanisms for handling variable argument lists conflict with the type substitution required for generic type parameters. This restriction also applies to the `params` keyword when used in combination with generic methods or methods within generic types.
- Supply the exact number of type arguments specified in the generic type or method declaration (**CS0305**). Each generic type parameter declared in the definition must have a corresponding type argument when the generic type is instantiated. The compiler needs to know which concrete type to substitute for each type parameter. For example, if a class is declared as `class MyList<T>`, you must provide exactly one type argument when using it, such as `MyList<int>`, not `MyList<int, string>`.
- Use only valid types as type arguments (**CS0306**). Pointer types, such as `int*` or `char*`, can't be used as type arguments because generic types require managed types that the garbage collector can track, and pointer types are unmanaged. If you need to work with pointers in a generic context, consider using `IntPtr` or restructuring your code to avoid mixing generics with unsafe code.
- Remove type argument syntax from non-generic constructs (**CS0307**, **CS0308**). Type arguments enclosed in angle brackets (like `<int>`) can only be applied to generic types and methods that declare type parameters. You must either remove the type arguments entirely or ensure you imported the namespace that contains the generic version of the type. For example, `IEnumerator<T>` requires the `using System.Collections.Generic;` directive, whereas `IEnumerator` is in `System.Collections`.

For more information, see Generic Type Parameters and Generics.

Constructor constraints

The following errors relate to the `new()` constraint on generic type parameters:

- **CS0304**: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- **CS0310**: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- **CS0417**: *'identifier': cannot provide arguments when creating an instance of a variable type.*

To correct these errors, add the `new()` constraint to type parameters that need to be instantiated, ensure type arguments have public parameterless constructors, and avoid passing arguments when constructing instances of type parameters:

- Add the `new()` constraint to the type parameter declaration (**CS0304**). When you use the `new` operator to create an instance of a type parameter within a generic type or method, the compiler must be able to guarantee that any type argument supplied at runtime has a parameterless constructor available. The `new()` constraint provides this guarantee at compile time, allowing the compiler to generate the appropriate instantiation code. For example, if you have `class C<T>` with a member `T t = new T();`, you must change the declaration to `class C<T> where T : new()`.
- Ensure that type arguments used with `new()` constrained type parameters have public parameterless constructors (**CS0310**). When a generic type or method declares a `new()` constraint on a type parameter, any concrete type used as a type argument must be non-abstract and must provide a public parameterless constructor. If a type only has non-public constructors (such as `private` or `protected`) or only has constructors with parameters, it can't satisfy the `new()` constraint. To fix this error, either add a public parameterless constructor to the type, or use a different type argument that already has one.
- Remove constructor arguments when instantiating type parameters (**CS0417**). The `new()` constraint only guarantees the existence of a parameterless constructor, so attempting to pass arguments to `new T(arguments)` isn't allowed because the compiler can't verify that a constructor with those specific parameter types exists on the type that are substituted for `T`. If you need to construct instances with specific arguments, consider using factory methods, abstract factory patterns, or specific base class/interface constraints that define the construction behavior you need.

For more information, see Constraints on type parameters and the new() constraint.

Constraint satisfaction and conversions

The following errors relate to type arguments not satisfying the constraints of generic type parameters:

- **CS0311**: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- **CS0312**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- **CS0313**: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- **CS0314**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- **CS0315**: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*

To correct these errors, use type arguments that satisfy all constraints through appropriate conversions, ensure derived classes repeat base class constraints, and understand that nullable value types have special constraint requirements:

- Change the type argument to one that has an implicit reference conversion to the constraint type (**CS0311**). When a type parameter has a constraint like `where T : BaseType`, any type argument must be convertible to `BaseType` through an implicit reference conversion or identity conversion. The type argument must either be `BaseType` itself, derive from `BaseType`, or implement `BaseType` if it's an interface. Implicit numeric conversions (such as from `short` to `int`) don't satisfy generic type parameter constraints because these conversions are value conversions, not reference conversions.
- Repeat the base class's type parameter constraints in any derived class declaration (**CS0314**). When a derived generic class inherits from a base generic class that has constraints on its type parameters, the derived class must declare the same constraints on its corresponding type parameters. Repetition is required because the compiler needs to verify that type arguments supplied to the derived class satisfies the requirements of the base class. For example, if you have `public class A<T> where T : SomeClass`, then any class deriving from it must be declared as `public class B<T> : A<T> where T : SomeClass`.
- Use non-nullable value types or change the constraint type (**CS0312**, **CS0313**). Nullable value types (such as `int?`) are distinct from their underlying value types and don't satisfy the same constraints. There's no implicit conversion between `int?` and `int`, and nullable

value types can't satisfy interface constraints because the nullable wrapper itself doesn't implement the interface, even though the underlying value type does. To fix these errors, either use the non-nullable form of the value type as the type argument, or adjust your constraint to accept `object` or a nullable reference type if appropriate.

- Ensure type arguments satisfy reference type or class constraints (**CS0315**). When a type parameter is constrained to a class type (such as `where T : SomeClass`), you can't use a value type (struct) as the type argument because there's no boxing conversion that satisfies the constraint relationship. The constraint requires a reference type that has an inheritance or implementation relationship with the constraint type. To resolve this error, either change the struct to a class if semantically appropriate, or remove the class constraint if the generic type can work with value types.

For more information, see Constraints on type parameters and Implicit conversions.

Generic type usage restrictions

The following errors relate to restrictions on how generic types can be used:

- **CS0403**: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- **CS0413**: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*
- **CS0695**: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- **CS0698**: *A generic type cannot derive from 'type' because it is an attribute class.*
- **CS9338**: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

To correct these errors, use `default` instead of `null` for unconstrained type parameters, add class constraints when using the `as` operator, avoid interface unification conflicts, don't create generic attribute classes, and ensure type arguments match the visibility of their containing members:

- Replace `null` assignments with `default(T)` or add a class constraint (**CS0403**). When you assign `null` to an unconstrained type parameter, the compiler can't guarantee that the type argument is a reference type that accepts `null` values, because it might be a value type like `int` or `struct`, which can't be `null`. To resolve this error, either use `default(T)`, which provides the appropriate default value for any type (null for reference types, zero or empty for value types), or add a `class` constraint to the type parameter if you specifically need reference type semantics and want to allow `null` assignments.
- Add a `class` or specific type constraint when using the `as` operator (**CS0413**). The `as` operator performs a safe type cast that returns `null` if the conversion fails, but this

behavior is incompatible with value types because value types can't be `null`. When you use `as` with an unconstrained type parameter, the compiler can't guarantee the type argument isn't a value type, so it rejects the code. To fix this error, add a `class` constraint or a specific reference type constraint (like `where T : SomeClass`) to ensure the type parameter is always a reference type that can properly handle the `null` result of a failed cast.

- Avoid implementing the same generic interface multiple times with type parameters that could unify (**CS0695**). When a class implements a generic interface multiple times with different type parameters (such as `class G<T1, T2> : I<T1>, I<T2>`), there's a risk that someone could instantiate it with the same type for both parameters (`G<int, int>`), which would create a conflict because the class would effectively be implementing `I<int>` twice. To resolve this error, either implement the interface only once, restructure your type parameters to prevent unification, or use separate non-generic classes for different specializations.
- Remove generic type parameters from attribute classes (**CS0698**).

  > **Note**
  >
  > This error is no longer produced in current versions of C#, as generic attributes are now supported.

- Ensure type arguments used in public or protected signatures are at least as accessible as the member using them (**CS9338**). A public or protected generic member must use type arguments that are publicly accessible. Otherwise external code couldn't properly reference or use the member's signature. For example, if you have `public class Container<T>` where `T` is an internal type, external assemblies can see the `Container` but can't properly work with it because they can't see `T`. To fix this error, either make the type argument public, or reduce the accessibility of the member using it to match the type argument's accessibility.

For more information, see Constraints on type parameters, default value expressions, and Attributes.

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

Last updated on 11/14/2025

Resolve errors and warnings related to generic type parameters and generic type arguments

This article covers the following compiler errors:

- CS0080: *Constraints are not allowed on non-generic declarations.*
- CS0081: *Type parameter declaration must be an identifier not a type.*
- CS0224: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- CS0304: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- CS0305: *Using the generic type 'generic type' requires 'number' type arguments.*
- CS0306: *The type 'type' may not be used as a type argument.*
- CS0307: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- CS0308: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*
- CS0310: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- CS0311: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- CS0312: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- CS0313: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- CS0314: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- CS0315: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*
- CS0403: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- CS0412: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- CS0413: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*

- CS0417: *'identifier': cannot provide arguments when creating an instance of a variable type.*
- CS0694: *Type parameter 'identifier' has the same name as the containing type, or method.*
- CS0695: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- CS0698: *A generic type cannot derive from 'type' because it is an attribute class.*
- CS9338: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

Type parameter declaration and naming

The following errors relate to how type parameters are declared and named in generic types and methods:

- **CS0080**: *Constraints are not allowed on non-generic declarations.*
- **CS0081**: *Type parameter declaration must be an identifier not a type.*
- **CS0412**: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- **CS0694**: *Type parameter 'identifier' has the same name as the containing type, or method.*

To correct these errors, ensure that type parameters are declared with valid identifiers, constraint clauses are only applied to generic declarations, and type parameter names don't conflict with other identifiers in scope:

- Remove the constraint clause from non-generic declarations (**CS0080**). The `where` clause can only be used on generic types and methods that declare type parameters, because constraints define requirements that type arguments must satisfy. If you need to apply constraints, first add type parameters to your type or method declaration. For example, change `public class MyClass where MyClass : System.IDisposable` to `public class MyClass<T> where T : System.IDisposable`.
- Replace actual type names with identifiers in type parameter declarations (**CS0081**). Type parameters must be declared using identifiers (like `T`, `TKey`, or `TValue`) rather than concrete types (like `int` or `string`), because the purpose of a type parameter is to serve as a placeholder that is substituted with actual types when the generic type or method is used. For example, change `public void F<int>()` to `public void F<T>()`.
- Rename type parameters, local variables, or parameters to avoid naming conflicts (**CS0412**, **CS0694**). Type parameter names can't shadow identifiers in the same scope. They can't match the name of the containing type or method. Such conflicts create ambiguity about which identifier is being referenced. For example, if you have a method `public void F<T>()`, you can't declare a local variable `double T` inside that method, and you can't name a type parameter the same as its containing type (`class C<C>`).

For more information, see Generic Type Parameters and Generics.

Type argument count and usage

The following errors relate to providing the correct number and type of type arguments to generic types and methods:

- **CS0224**: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- **CS0305**: *Using the generic type 'generic type' requires 'number' type arguments.*
- **CS0306**: *The type 'type' may not be used as a type argument.*
- **CS0307**: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- **CS0308**: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*

To correct these errors, ensure that you provide the exact number of type arguments required by the generic declaration. Use only valid types as type arguments. Don't apply type arguments to non-generic constructs:

- Remove generic type parameters or containing generic type declarations from methods that use `__arglist` (**CS0224**). The `__arglist` keyword is incompatible with generics because the runtime mechanisms for handling variable argument lists conflict with the type substitution required for generic type parameters. This restriction also applies to the `params` keyword when used in combination with generic methods or methods within generic types.
- Supply the exact number of type arguments specified in the generic type or method declaration (**CS0305**). Each generic type parameter declared in the definition must have a corresponding type argument when the generic type is instantiated. The compiler needs to know which concrete type to substitute for each type parameter. For example, if a class is declared as `class MyList<T>`, you must provide exactly one type argument when using it, such as `MyList<int>`, not `MyList<int, string>`.
- Use only valid types as type arguments (**CS0306**). Pointer types, such as `int*` or `char*`, can't be used as type arguments because generic types require managed types that the garbage collector can track, and pointer types are unmanaged. If you need to work with pointers in a generic context, consider using `IntPtr` or restructuring your code to avoid mixing generics with unsafe code.
- Remove type argument syntax from non-generic constructs (**CS0307**, **CS0308**). Type arguments enclosed in angle brackets (like `<int>`) can only be applied to generic types and methods that declare type parameters. You must either remove the type arguments entirely or ensure you imported the namespace that contains the generic version of the type. For example, `IEnumerator<T>` requires the `using System.Collections.Generic;` directive, whereas `IEnumerator` is in `System.Collections`.

For more information, see Generic Type Parameters and Generics.

Constructor constraints

The following errors relate to the `new()` constraint on generic type parameters:

- **CS0304**: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- **CS0310**: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- **CS0417**: *'identifier': cannot provide arguments when creating an instance of a variable type.*

To correct these errors, add the `new()` constraint to type parameters that need to be instantiated, ensure type arguments have public parameterless constructors, and avoid passing arguments when constructing instances of type parameters:

- Add the `new()` constraint to the type parameter declaration (**CS0304**). When you use the `new` operator to create an instance of a type parameter within a generic type or method, the compiler must be able to guarantee that any type argument supplied at runtime has a parameterless constructor available. The `new()` constraint provides this guarantee at compile time, allowing the compiler to generate the appropriate instantiation code. For example, if you have `class C<T>` with a member `T t = new T();`, you must change the declaration to `class C<T> where T : new()`.
- Ensure that type arguments used with `new()` constrained type parameters have public parameterless constructors (**CS0310**). When a generic type or method declares a `new()` constraint on a type parameter, any concrete type used as a type argument must be non-abstract and must provide a public parameterless constructor. If a type only has non-public constructors (such as `private` or `protected`) or only has constructors with parameters, it can't satisfy the `new()` constraint. To fix this error, either add a public parameterless constructor to the type, or use a different type argument that already has one.
- Remove constructor arguments when instantiating type parameters (**CS0417**). The `new()` constraint only guarantees the existence of a parameterless constructor, so attempting to pass arguments to `new T(arguments)` isn't allowed because the compiler can't verify that a constructor with those specific parameter types exists on the type that are substituted for `T`. If you need to construct instances with specific arguments, consider using factory methods, abstract factory patterns, or specific base class/interface constraints that define the construction behavior you need.

For more information, see Constraints on type parameters and the new() constraint.

Constraint satisfaction and conversions

The following errors relate to type arguments not satisfying the constraints of generic type parameters:

- **CS0311**: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- **CS0312**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- **CS0313**: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- **CS0314**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- **CS0315**: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*

To correct these errors, use type arguments that satisfy all constraints through appropriate conversions, ensure derived classes repeat base class constraints, and understand that nullable value types have special constraint requirements:

- Change the type argument to one that has an implicit reference conversion to the constraint type (**CS0311**). When a type parameter has a constraint like `where T : BaseType`, any type argument must be convertible to `BaseType` through an implicit reference conversion or identity conversion. The type argument must either be `BaseType` itself, derive from `BaseType`, or implement `BaseType` if it's an interface. Implicit numeric conversions (such as from `short` to `int`) don't satisfy generic type parameter constraints because these conversions are value conversions, not reference conversions.
- Repeat the base class's type parameter constraints in any derived class declaration (**CS0314**). When a derived generic class inherits from a base generic class that has constraints on its type parameters, the derived class must declare the same constraints on its corresponding type parameters. Repetition is required because the compiler needs to verify that type arguments supplied to the derived class satisfies the requirements of the base class. For example, if you have `public class A<T> where T : SomeClass`, then any class deriving from it must be declared as `public class B<T> : A<T> where T : SomeClass`.
- Use non-nullable value types or change the constraint type (**CS0312**, **CS0313**). Nullable value types (such as `int?`) are distinct from their underlying value types and don't satisfy the same constraints. There's no implicit conversion between `int?` and `int`, and nullable

value types can't satisfy interface constraints because the nullable wrapper itself doesn't implement the interface, even though the underlying value type does. To fix these errors, either use the non-nullable form of the value type as the type argument, or adjust your constraint to accept `object` or a nullable reference type if appropriate.

- Ensure type arguments satisfy reference type or class constraints (**CS0315**). When a type parameter is constrained to a class type (such as `where T : SomeClass`), you can't use a value type (struct) as the type argument because there's no boxing conversion that satisfies the constraint relationship. The constraint requires a reference type that has an inheritance or implementation relationship with the constraint type. To resolve this error, either change the struct to a class if semantically appropriate, or remove the class constraint if the generic type can work with value types.

For more information, see Constraints on type parameters and Implicit conversions.

Generic type usage restrictions

The following errors relate to restrictions on how generic types can be used:

- **CS0403**: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- **CS0413**: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*
- **CS0695**: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- **CS0698**: *A generic type cannot derive from 'type' because it is an attribute class.*
- **CS9338**: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

To correct these errors, use `default` instead of `null` for unconstrained type parameters, add class constraints when using the `as` operator, avoid interface unification conflicts, don't create generic attribute classes, and ensure type arguments match the visibility of their containing members:

- Replace `null` assignments with `default(T)` or add a class constraint (**CS0403**). When you assign `null` to an unconstrained type parameter, the compiler can't guarantee that the type argument is a reference type that accepts `null` values, because it might be a value type like `int` or `struct`, which can't be `null`. To resolve this error, either use `default(T)`, which provides the appropriate default value for any type (null for reference types, zero or empty for value types), or add a `class` constraint to the type parameter if you specifically need reference type semantics and want to allow `null` assignments.
- Add a `class` or specific type constraint when using the `as` operator (**CS0413**). The `as` operator performs a safe type cast that returns `null` if the conversion fails, but this

behavior is incompatible with value types because value types can't be `null`. When you use `as` with an unconstrained type parameter, the compiler can't guarantee the type argument isn't a value type, so it rejects the code. To fix this error, add a `class` constraint or a specific reference type constraint (like `where T : SomeClass`) to ensure the type parameter is always a reference type that can properly handle the `null` result of a failed cast.

- Avoid implementing the same generic interface multiple times with type parameters that could unify (**CS0695**). When a class implements a generic interface multiple times with different type parameters (such as `class G<T1, T2> : I<T1>, I<T2>`), there's a risk that someone could instantiate it with the same type for both parameters (`G<int, int>`), which would create a conflict because the class would effectively be implementing `I<int>` twice. To resolve this error, either implement the interface only once, restructure your type parameters to prevent unification, or use separate non-generic classes for different specializations.
- Remove generic type parameters from attribute classes (**CS0698**).

  > **Note**
  >
  > This error is no longer produced in current versions of C#, as generic attributes are now supported.

- Ensure type arguments used in public or protected signatures are at least as accessible as the member using them (**CS9338**). A public or protected generic member must use type arguments that are publicly accessible. Otherwise external code couldn't properly reference or use the member's signature. For example, if you have `public class Container<T>` where `T` is an internal type, external assemblies can see the `Container` but can't properly work with it because they can't see `T`. To fix this error, either make the type argument public, or reduce the accessibility of the member using it to match the type argument's accessibility.

For more information, see Constraints on type parameters, default value expressions, and Attributes.

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

Last updated on 11/14/2025

Resolve errors and warnings related to generic type parameters and generic type arguments

This article covers the following compiler errors:

- CS0080: *Constraints are not allowed on non-generic declarations.*
- CS0081: *Type parameter declaration must be an identifier not a type.*
- CS0224: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- CS0304: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- CS0305: *Using the generic type 'generic type' requires 'number' type arguments.*
- CS0306: *The type 'type' may not be used as a type argument.*
- CS0307: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- CS0308: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*
- CS0310: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- CS0311: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- CS0312: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- CS0313: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- CS0314: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- CS0315: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*
- CS0403: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- CS0412: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- CS0413: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*

- CS0417: *'identifier': cannot provide arguments when creating an instance of a variable type.*
- CS0694: *Type parameter 'identifier' has the same name as the containing type, or method.*
- CS0695: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- CS0698: *A generic type cannot derive from 'type' because it is an attribute class.*
- CS9338: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

Type parameter declaration and naming

The following errors relate to how type parameters are declared and named in generic types and methods:

- **CS0080**: *Constraints are not allowed on non-generic declarations.*
- **CS0081**: *Type parameter declaration must be an identifier not a type.*
- **CS0412**: *'parameter': a parameter, local variable, or local function cannot have the same name as a method type parameter.*
- **CS0694**: *Type parameter 'identifier' has the same name as the containing type, or method.*

To correct these errors, ensure that type parameters are declared with valid identifiers, constraint clauses are only applied to generic declarations, and type parameter names don't conflict with other identifiers in scope:

- Remove the constraint clause from non-generic declarations (**CS0080**). The `where` clause can only be used on generic types and methods that declare type parameters, because constraints define requirements that type arguments must satisfy. If you need to apply constraints, first add type parameters to your type or method declaration. For example, change `public class MyClass where MyClass : System.IDisposable` to `public class MyClass<T> where T : System.IDisposable`.
- Replace actual type names with identifiers in type parameter declarations (**CS0081**). Type parameters must be declared using identifiers (like `T`, `TKey`, or `TValue`) rather than concrete types (like `int` or `string`), because the purpose of a type parameter is to serve as a placeholder that is substituted with actual types when the generic type or method is used. For example, change `public void F<int>()` to `public void F<T>()`.
- Rename type parameters, local variables, or parameters to avoid naming conflicts (**CS0412**, **CS0694**). Type parameter names can't shadow identifiers in the same scope. They can't match the name of the containing type or method. Such conflicts create ambiguity about which identifier is being referenced. For example, if you have a method `public void F<T>()`, you can't declare a local variable `double T` inside that method, and you can't name a type parameter the same as its containing type (`class C<C>`).

For more information, see Generic Type Parameters and Generics.

Type argument count and usage

The following errors relate to providing the correct number and type of type arguments to generic types and methods:

- **CS0224**: *A method with vararg cannot be generic, be in a generic type, or have a params parameter.*
- **CS0305**: *Using the generic type 'generic type' requires 'number' type arguments.*
- **CS0306**: *The type 'type' may not be used as a type argument.*
- **CS0307**: *The 'construct' 'identifier' is not a generic method. If you intended an expression list, use parentheses around the < expression.*
- **CS0308**: *The non-generic type-or-method 'identifier' cannot be used with type arguments.*

To correct these errors, ensure that you provide the exact number of type arguments required by the generic declaration. Use only valid types as type arguments. Don't apply type arguments to non-generic constructs:

- Remove generic type parameters or containing generic type declarations from methods that use `__arglist` (**CS0224**). The `__arglist` keyword is incompatible with generics because the runtime mechanisms for handling variable argument lists conflict with the type substitution required for generic type parameters. This restriction also applies to the `params` keyword when used in combination with generic methods or methods within generic types.
- Supply the exact number of type arguments specified in the generic type or method declaration (**CS0305**). Each generic type parameter declared in the definition must have a corresponding type argument when the generic type is instantiated. The compiler needs to know which concrete type to substitute for each type parameter. For example, if a class is declared as `class MyList<T>`, you must provide exactly one type argument when using it, such as `MyList<int>`, not `MyList<int, string>`.
- Use only valid types as type arguments (**CS0306**). Pointer types, such as `int*` or `char*`, can't be used as type arguments because generic types require managed types that the garbage collector can track, and pointer types are unmanaged. If you need to work with pointers in a generic context, consider using `IntPtr` or restructuring your code to avoid mixing generics with unsafe code.
- Remove type argument syntax from non-generic constructs (**CS0307**, **CS0308**). Type arguments enclosed in angle brackets (like `<int>`) can only be applied to generic types and methods that declare type parameters. You must either remove the type arguments entirely or ensure you imported the namespace that contains the generic version of the type. For example, `IEnumerator<T>` requires the `using System.Collections.Generic;` directive, whereas `IEnumerator` is in `System.Collections`.

For more information, see Generic Type Parameters and Generics.

Constructor constraints

The following errors relate to the `new()` constraint on generic type parameters:

- **CS0304**: *Cannot create an instance of the variable type 'type' because it does not have the new() constraint.*
- **CS0310**: *The type 'typename' must be a non-abstract type with a public parameterless constructor in order to use it as parameter 'parameter' in the generic type or method 'generic'.*
- **CS0417**: *'identifier': cannot provide arguments when creating an instance of a variable type.*

To correct these errors, add the `new()` constraint to type parameters that need to be instantiated, ensure type arguments have public parameterless constructors, and avoid passing arguments when constructing instances of type parameters:

- Add the `new()` constraint to the type parameter declaration (**CS0304**). When you use the `new` operator to create an instance of a type parameter within a generic type or method, the compiler must be able to guarantee that any type argument supplied at runtime has a parameterless constructor available. The `new()` constraint provides this guarantee at compile time, allowing the compiler to generate the appropriate instantiation code. For example, if you have `class C<T>` with a member `T t = new T();`, you must change the declaration to `class C<T> where T : new()`.
- Ensure that type arguments used with `new()` constrained type parameters have public parameterless constructors (**CS0310**). When a generic type or method declares a `new()` constraint on a type parameter, any concrete type used as a type argument must be non-abstract and must provide a public parameterless constructor. If a type only has non-public constructors (such as `private` or `protected`) or only has constructors with parameters, it can't satisfy the `new()` constraint. To fix this error, either add a public parameterless constructor to the type, or use a different type argument that already has one.
- Remove constructor arguments when instantiating type parameters (**CS0417**). The `new()` constraint only guarantees the existence of a parameterless constructor, so attempting to pass arguments to `new T(arguments)` isn't allowed because the compiler can't verify that a constructor with those specific parameter types exists on the type that are substituted for `T`. If you need to construct instances with specific arguments, consider using factory methods, abstract factory patterns, or specific base class/interface constraints that define the construction behavior you need.

For more information, see Constraints on type parameters and the new() constraint.

Constraint satisfaction and conversions

The following errors relate to type arguments not satisfying the constraints of generic type parameters:

- **CS0311**: *The type 'type1' cannot be used as type parameter 'T' in the generic type or method '<name>'. There is no implicit reference conversion from 'type1' to 'type2'.*
- **CS0312**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. The nullable type 'type1' does not satisfy the constraint of 'type2'.*
- **CS0313**: *The type 'type1' cannot be used as type parameter 'parameter name' in the generic type or method 'type2'. The nullable type 'type1' does not satisfy the constraint of 'type2'. Nullable types can not satisfy any interface constraints.*
- **CS0314**: *The type 'type1' cannot be used as type parameter 'name' in the generic type or method 'name'. There is no boxing conversion or type parameter conversion from 'type1' to 'type2'.*
- **CS0315**: *The type 'valueType' cannot be used as type parameter 'T' in the generic type or method 'TypeorMethod<T>'. There is no boxing conversion from 'valueType' to 'referenceType'.*

To correct these errors, use type arguments that satisfy all constraints through appropriate conversions, ensure derived classes repeat base class constraints, and understand that nullable value types have special constraint requirements:

- Change the type argument to one that has an implicit reference conversion to the constraint type (**CS0311**). When a type parameter has a constraint like `where T : BaseType`, any type argument must be convertible to `BaseType` through an implicit reference conversion or identity conversion. The type argument must either be `BaseType` itself, derive from `BaseType`, or implement `BaseType` if it's an interface. Implicit numeric conversions (such as from `short` to `int`) don't satisfy generic type parameter constraints because these conversions are value conversions, not reference conversions.
- Repeat the base class's type parameter constraints in any derived class declaration (**CS0314**). When a derived generic class inherits from a base generic class that has constraints on its type parameters, the derived class must declare the same constraints on its corresponding type parameters. Repetition is required because the compiler needs to verify that type arguments supplied to the derived class satisfies the requirements of the base class. For example, if you have `public class A<T> where T : SomeClass`, then any class deriving from it must be declared as `public class B<T> : A<T> where T : SomeClass`.
- Use non-nullable value types or change the constraint type (**CS0312**, **CS0313**). Nullable value types (such as `int?`) are distinct from their underlying value types and don't satisfy the same constraints. There's no implicit conversion between `int?` and `int`, and nullable

value types can't satisfy interface constraints because the nullable wrapper itself doesn't implement the interface, even though the underlying value type does. To fix these errors, either use the non-nullable form of the value type as the type argument, or adjust your constraint to accept `object` or a nullable reference type if appropriate.

- Ensure type arguments satisfy reference type or class constraints (**CS0315**). When a type parameter is constrained to a class type (such as `where T : SomeClass`), you can't use a value type (struct) as the type argument because there's no boxing conversion that satisfies the constraint relationship. The constraint requires a reference type that has an inheritance or implementation relationship with the constraint type. To resolve this error, either change the struct to a class if semantically appropriate, or remove the class constraint if the generic type can work with value types.

For more information, see Constraints on type parameters and Implicit conversions.

Generic type usage restrictions

The following errors relate to restrictions on how generic types can be used:

- **CS0403**: *Cannot convert null to type parameter 'name' because it could be a non-nullable value type. Consider using 'default('T')' instead.*
- **CS0413**: *The type parameter 'type parameter' cannot be used with the 'as' operator because it does not have a class type constraint nor a 'class' constraint.*
- **CS0695**: *'type' cannot implement both 'interface1' and 'interface2' because they may unify for some type parameter substitutions.*
- **CS0698**: *A generic type cannot derive from 'type' because it is an attribute class.*
- **CS9338**: *Inconsistent accessibility: type 'type1' is less accessible than class 'type2'.*

To correct these errors, use `default` instead of `null` for unconstrained type parameters, add class constraints when using the `as` operator, avoid interface unification conflicts, don't create generic attribute classes, and ensure type arguments match the visibility of their containing members:

- Replace `null` assignments with `default(T)` or add a class constraint (**CS0403**). When you assign `null` to an unconstrained type parameter, the compiler can't guarantee that the type argument is a reference type that accepts `null` values, because it might be a value type like `int` or `struct`, which can't be `null`. To resolve this error, either use `default(T)`, which provides the appropriate default value for any type (null for reference types, zero or empty for value types), or add a `class` constraint to the type parameter if you specifically need reference type semantics and want to allow `null` assignments.
- Add a `class` or specific type constraint when using the `as` operator (**CS0413**). The `as` operator performs a safe type cast that returns `null` if the conversion fails, but this

behavior is incompatible with value types because value types can't be `null`. When you use `as` with an unconstrained type parameter, the compiler can't guarantee the type argument isn't a value type, so it rejects the code. To fix this error, add a `class` constraint or a specific reference type constraint (like `where T : SomeClass`) to ensure the type parameter is always a reference type that can properly handle the `null` result of a failed cast.

- Avoid implementing the same generic interface multiple times with type parameters that could unify (**CS0695**). When a class implements a generic interface multiple times with different type parameters (such as `class G<T1, T2> : I<T1>, I<T2>`), there's a risk that someone could instantiate it with the same type for both parameters (`G<int, int>`), which would create a conflict because the class would effectively be implementing `I<int>` twice. To resolve this error, either implement the interface only once, restructure your type parameters to prevent unification, or use separate non-generic classes for different specializations.
- Remove generic type parameters from attribute classes (**CS0698**).

  > **Note**
  >
  > This error is no longer produced in current versions of C#, as generic attributes are now supported.

- Ensure type arguments used in public or protected signatures are at least as accessible as the member using them (**CS9338**). A public or protected generic member must use type arguments that are publicly accessible. Otherwise external code couldn't properly reference or use the member's signature. For example, if you have `public class Container<T>` where `T` is an internal type, external assemblies can see the `Container` but can't properly work with it because they can't see `T`. To fix this error, either make the type argument public, or reduce the accessibility of the member using it to match the type argument's accessibility.

For more information, see Constraints on type parameters, default value expressions, and Attributes.

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

Last updated on 11/14/2025

Compiler Error CS0699

Article • 09/15/2021

'generic' does not define type parameter 'identifier'

A type parameter was used in a generic definition that was not found in the declaration list of the type parameters for that generic. This can happen if the name used for the type parameter was inconsistent.

The following sample generates CS0699:

C#

```
// CS0699.cs
class C<T> where U : I   // CS0699 – U is not a valid type parameter
{
}
```

Compiler Error CS0701

Article • 09/15/2021

'identifier' is not a valid constraint. A type used as a constraint must be an interface, a non-sealed class or a type parameter.

This error occurs if a sealed type is used as a constraint. To resolve this error, use only non-sealed types as constraints.

Example

The following sample generates CS0701.

C#

```
// CS0701.cs
// compile with: /target:library
class C<T> where T : System.String {}   // CS0701
class D<T> where T : System.Attribute {}   // OK
```

Compiler error CS0702

Article • 09/15/2021

Constraint cannot be special class 'identifier'

The following types may not be used as constraints: Object, Array, or ValueType.

Example

The following sample generates CS0702:

C#

```
// CS0702.cs
class C<T> where T : System.Array  // CS0702
{
}
```

See also

- Constraints on Type Parameters

Compiler Error CS0703

Article • 09/15/2021

Inconsistent accessibility: constraint type 'identifier' is less accessible than 'identifier'

A constraint may not force the generic parameter to be less accessible than the generic class itself. In the following example, while the generic class C<T> is declared public, the constraint attempts to force T to implement an internal interface. Even if this were allowed, only clients with internal access would be able to create the parameter for the class, so in effect, the class could be used only by clients with internal access.

To get rid of this error, make sure the access level of the generic class is not less restrictive than any classes or interfaces that appear in the bounds.

The following sample generates CS0703:

C#

```
// CS0703.cs
internal interface I {}
public class C<T> where T : I  // CS0703 - I is internal; C<T> is public
{
}
```

Compiler Error CS0704

Article • 09/15/2021

Cannot do member lookup in 'type' because it is a type parameter

An inner type cannot be specified through a type parameter. Try using the desired type explicitly.

Example

The following sample generates CS0704:

C#

```
// CS0704.cs
class B
{
    public class I { }
}

class C<T> where T : B
{
    T.I f;  // CS0704 - member lookup on type parameter
    // Try this instead:
    // B.I f;
}

class CMain
{
    public static void Main() {}
}
```

Compiler Error CS0706

Article • 09/15/2021

Invalid constraint type. A type used as a constraint must be an interface, a non-sealed class or a type parameter.

This error occurs when an invalid construct is used in a constraint clause. To avoid this error, use an interface or non-sealed class instead of the construct that caused the error.

Example

The following sample generates CS0706.

C#

```
// CS0706.cs
// compile with: /target:library
class A {}
class C<T> where T : int[] {}  // CS0706
class D<T> where T : A {}  // OK
```

Compiler Error CS0708

Article • 09/15/2021

'field': cannot declare instance members in a static class

This error occurs if you declare a non-static member in a class that is declared static. It is not possible to create instances of static classes, so instance variables would not be meaningful. The **static** keyword should be applied to all members of static classes.

The following sample generates CS0708:

C#

```
// CS0708.cs
// compile with: /target:library
public static class C
{
   int i;  // CS0708
   static int j;  // OK
}
```

Compiler Error CS0709

Article • 09/15/2021

'derived class': cannot derive from static class 'base class'

A static class cannot be instantiated or derived from. That is, a static class cannot be a base class of any other class.

Example

The following sample generates CS0709.

C#

```
// CS0709.cs
// compile with: /target:library
public static class Base {}
public class Derived : Base {}   // CS0709
```

Compiler Error CS0711

Article • 09/16/2021

Static classes cannot contain destructors

A static class cannot be instantiated, hence it has no need for constructors or finalizer. To avoid this error, remove finalizer from static classes, or, if you really want to construct and destroy instances, make the class non-static.

The following sample generates CS0711:

C#

```
// CS0711.cs
public static class C
{
   ~C()  // CS0711
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0712

Article • 09/15/2021

Cannot create an instance of the static class 'static class'

It is not possible to create instances of static classes. Static classes are designed to contain static fields and methods, but may not be instantiated.

Example

The following sample generates CS0712:

C#

```
// CS0712.cs
public static class SC
{
}

public class CMain
{
    public static void Main()
    {
        SC sc = new SC();  // CS0712
    }
}
```

Compiler Error CS0713

Article • 09/15/2021

Static class 'static type' cannot derive from type 'type'. Static classes must derive from object.

If this were allowed, the static class would inherit methods and non-static members from the base class, so it would not be static. Therefore, it is not allowed.

The following sample generates CS0713:

C#

```
// CS0713.cs
public class Base
{
}

public static class Derived : Base  // CS0713
{
}

public class CMain
{
   public static void Main()
   {
   }
}
```

Compiler Error CS0714

Article • 09/15/2021

'static type' : static classes cannot implement interfaces

Interfaces may define non-static methods on objects and hence may not be implemented by static classes. To resolve this error, make sure your class does not attempt to implement any interfaces.

Example

The following sample generates CS0714:

C#

```
// CS0714.cs
interface I
{
}

public static class C : I  // CS0714
{
    public static void Main()
    {
    }
}
```

Compiler Error CS0716

Article • 09/15/2021

Cannot convert to static type 'type'

This error occurs if your code uses a cast to convert to a static type. Since it is not possible for an object to be an instance of a static type, casting to a static type can never be a meaningful cast.

Example

The following sample generates CS0716:

C#

```
// CS0716.cs

public static class SC
{
    static void F() { }
}

public class Test
{
    public static void Main()
    {
        object o = new object();
        System.Console.WriteLine((SC)o);  // CS0716
    }
}
```

Compiler Error CS0717

Article • 09/15/2021

'static class': static classes cannot be used as constraints

Static classes cannot be extended as they only contain static members and not instance members. Because they cannot be extended, this makes them useless as type parameters and constraints, as no type can exist that is a specialization of a static class.

Example

The following sample generates CS0717:

C#

```
// CS0717.cs

public static class SC
{
    public static void F()
    {
    }
}

public class G<T> where T : SC  // CS0717
{
    public static void Main()
    {
    }
}
```

Compiler Error CS0718

Article • 09/15/2021

'type': static types cannot be used as type arguments

Because a static type cannot be instantiated, it cannot be used as a generic argument. To resolve this error, remove the static type from the generic argument.

Example

The following sample generates CS0718:

C#

```
// CS0718.cs
public static class SC
{
    public static void F()
    {
    }
}

public class G<T>
{
}

public class CMain
{
    public static void Main()
    {
        G<SC> gsc = new G<SC>();  // CS0718
    }
}
```

Compiler Error CS0720

Article • 09/15/2021

'static class': cannot declare indexers in a static class

Indexers are not meaningful in static classes, since they can only be used with instances, and it is not possible to create instances of a static type.

Example

The following sample generates CS0720:

C#

```
// CS0720.cs

public static class Test
{
    public int this[int index]  // CS0720
    {
        get { return 1; }
        set {}
    }

    static void Main() {}
}
```

Compiler Error CS0721

Article • 09/15/2021

'type': static types cannot be used as parameters

A static type is not meaningful as a parameter. Since no instances of static types may be created, no instance could ever be passed as a parameter.

The following sample generates CS0721:

C#

```
// CS0721.cs
public static class SC
{
}

public class CMain
{
   public void F(SC sc)  // CS0721
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0722

Article • 09/15/2021

'type': static types cannot be used as return types

A static type as a return type is not meaningful since instances of static types cannot be created.

The following sample generates CS0722:

C#

```
// CS0722.cs
public static class SC
{
}

public class CMain
{
   public SC F()  // CS0722
   {
      return null;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS0723

Article • 09/15/2021

Cannot declare variable of static type 'type'

Instances of static types cannot be created.

The following sample generates CS0723:

C#

```
// CS0723.cs
public static class SC
{
}

public class CMain
{
   public static void Main()
   {
      SC sc = null;  // CS0723
   }
}
```

Compiler Error CS0724

Article • 09/15/2021

A throw statement with no arguments is not allowed in a finally clause that is nested inside the nearest enclosing catch clause

Example

The following example generates CS0724 because of the `throw` statement inside the `finally` clause block:

C#

```
// CS0724.cs
using System;

class Program
{
    static void Test()
    {
        try
        {
            throw new Exception();
        }
        catch
        {
            try
            {
            }
            finally
            {
                throw; // CS0724
            }
        }
    }

    static void Main()
    {
    }
}
```

Compiler Error CS0726

Article • 09/15/2021

'format specifier' is not a valid format specifier

This error occurs in the debugger. When you type a variable name into one of the debugger windows, you can follow it with a comma, and then a format specifier. Examples are: `myInt, h` or `myString,nq`. This error arises when the compiler does not recognize the Format Specifiers in C#.

Compiler Error CS0727

Article • 09/15/2021

Invalid format specifier

This error occurs in the debugger. When you type a variable name into one of the debugger windows, you can follow it with a comma, and then a format specifier. Examples are: myInt, h; or myString,nq. This error arises when the compiler is completely unable to parse what you typed in. To resolve this error, retype the variable name, optionally followed by a comma and a valid Format Specifier.

Compiler Error CS0729

Article • 09/15/2021

Type 'type' is defined in this assembly, but a type forwarder is specified for it

You cannot use a type forwarder for a type defined in the same assembly.

Example

The following sample generates CS0729.

C#

```
// CS0729.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:TypeForwardedTo(typeof(TestClass))]   // CS0729
class TestClass {}
```

Compiler Error CS0730

Article • 09/15/2021

Cannot forward type 'type' because it is a nested type of 'type'

This error is generated when you try to forward a nested class.

Example

The following sample generates CS0730. It consists of two source files. First, compile the library file `CS0730a.cs`, and the compile the file `CS0730.cs` referencing the library file.

C#

```
// CS0730a.cs
// compile with: /t:library
public class Outer
{
   public class Nested {}
}
```

C#

```
// CS0730.cs
// compile with: /t:library /r:CS0730a.dll
using System.Runtime.CompilerServices;

[assembly:TypeForwardedToAttribute(typeof(Outer.Nested))]   // CS0730

[assembly:TypeForwardedToAttribute(typeof(Outer))]   // OK
```

Compiler Error CS0731

Article • 09/15/2021

The type forwarder for type 'type' in assembly 'assembly' causes a cycle

This error can only occur with improperly formed imported metadata. It cannot occur with only C# source.

Example

The following sample generates CS0731. The example consists of three files:

1. Circular.IL
2. Circular2.IL
3. CS0731.cs

First compile the .IL files as libraries, and then compile the .cs code referencing the two files.

il

```
// Circular.il
// compile with: /DLL /out=Circular.dll
.assembly extern circular2
{
  .ver 0:0:0:0
}
.assembly extern circular3
{
  .ver 0:0:0:0
}
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )   // .z\V.4..
  .ver 2:0:0:0
}
.assembly Circular
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.
ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.class extern forwarder Circular.Referenced.TypeForwarder
{
```

```
  .assembly extern circular2
}
.module Circular.dll
// MVID: {880C2329-C915-42A0-83E9-9D10C3E6DBD0}
.imagebase 0x00400000
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       // WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x04E40000
// ======== CLASS MEMBERS DECLARATION =========
.class public abstract auto ansi sealed beforefieldinit User
       extends [mscorlib]System.Object
{
  .method public hidebysig static class
[circular2]Circular.Referenced.TypeForwarder
          F() cil managed
  {
    .maxstack  1
    newobj     instance void
[circular2]Circular.Referenced.TypeForwarder::.ctor()
    ret
  }
}
```

il

```
// Circular2.il
// compile with: /DLL /out=Circular2.dll
.assembly extern mscorlib
{
  .publickeytoken = (B7 7A 5C 56 19 34 E0 89 )   // .z\V.4..
  .ver 2:0:0:0
}
.assembly extern Circular
{
  .ver 0:0:0:0
}
.assembly circular2
{
  .custom instance void
[mscorlib]System.Runtime.CompilerServices.CompilationRelaxationsAttribute::.
ctor(int32) = ( 01 00 08 00 00 00 00 00 )
  .hash algorithm 0x00008004
  .ver 0:0:0:0
}
.class extern forwarder Circular.Referenced.TypeForwarder
{
  .assembly extern Circular
}
.module circular2.dll
// MVID: {8B3BE5C8-DBE1-49C4-BC72-DF35F0387C21}
.imagebase 0x00400000
```

```
.file alignment 0x00000200
.stackreserve 0x00100000
.subsystem 0x0003       //   WINDOWS_CUI
.corflags 0x00000001    //  ILONLY
// Image base: 0x04E40000
```

C#

```
// CS0731.cs
// compile with: /reference:circular.dll /reference:circular2.dll
// CS0731 expected
class A {
   public static void Main() {
      User.F();
   }
}
```

Compiler Error CS0733

Article • 09/15/2021

Cannot forward generic type, 'GenericType<>'

Example

The following example generates CS0733. Compile the first file as a library, and then reference it when you compile the second file.

C#

```
// CS0733a.cs
// compile with: /target:library
public class GenericType<T>
{
}
```

C#

```
// CS0733.cs
// compile with: /target:library /r:CS0733a.dll
[assembly:
System.Runtime.CompilerServices.TypeForwardedTo(typeof(GenericType<int>))]
// CS0733
```

Compiler Error CS0734

08/08/2025

The /moduleassemblyname option may only be specified when building a target type of 'module'

The compiler option **ModuleAssemblyName** should only be used when building a .netmodule. See **ModuleAssemblyName** (C# Compiler Option) for more information.

For more information on building a .netmodule, see the **module** element for the **OutputType** (C# Compiler Options).

Example

The following sample generates CS0734. To resolve, add `<TargetType>module</TargetType>` to the compilation.

C#

```
// CS0734.cs
// compile with: /moduleassemblyname:A
// CS0734 expected
public class Test {}
```

Compiler Error CS0735

Article • 09/15/2021

Invalid type specified as an argument for TypeForwardedTo attribute

The following sample generates CS0735.

C#

```
// CS735.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:TypeForwardedTo(typeof(int[]))]   // CS0735
[assembly:TypeForwardedTo(typeof(string))]   // OK
```

Compiler Error CS0739

Article • 09/15/2021

'type name' duplicate TypeForwardedToAttribute.

An assembly can have no more than one TypeForwardedToAttribute to an external type.

To correct this error

1. Locate and remove the duplicate TypeForwardedToAttribute.

Example

The following code generates CS0739:

C#

```
// CS0739.cs
// CS0739
// Assume that a class Test is declared in a separate dll
// with a namespace that is named cs739dll.
[assembly:
System.Runtime.CompilerServices.TypeForwardedTo(typeof(cs739dll.Test))]
[assembly:
System.Runtime.CompilerServices.TypeForwardedTo(typeof(cs739dll.Test))]
namespace cs0739
{
    class Program
    {
        static void Main(string[] args)
        {
        }
    }
}
```

See also

- TypeForwardedToAttribute

Compiler Error CS0742

Article • 09/15/2021

A query body must end with a select clause or a group clause

A query expression must terminate with either a `select` clause or a `group` clause without a continuation.

To correct this error

1. Add a select clause or group clause to the query.

Example

The following code generates CS0742:

```
C#
```

```
// cs0742.cs
using System.Linq;
public class Test
{
    public static int Main()
    {
        int[] array = { 1, 2, 3 };
        var query = from num in array; // CS0742
        return 1;
    }
}
```

If the `group` clause uses the into keyword to store the results of the grouping into a temporary identifier, it cannot be the last clause in a query. A `select` or second `group` clause is still required.

See also

- LINQ Query Expressions

Compiler Error CS0743

Article • 09/15/2021

Expected contextual keyword 'on'

The pattern for a `join` clause is `join` ... `in` ... `on` ... `equals`, as shown in this example:

```
C#
```

```
var query = from x in array1
            join y in array2 on x equals y
            select x;
```

To correct this error

1. Add the `on` keyword to the `join` clause.

Example

The following code generates CS0743:

```
C#
```

```
// cs0743.cs
using System;
using System.Linq;

public class C
{
    public static int Main()
    {
        int[] array1 = { 1, 2, 3, 4, 5, 6 };
        int[] array2 = { 5, 6, 7, 8, 9 };
        var c = from x in array1
                join y in array2 x equals y // CS0743
                select x;
        return 1;
    }
}
```

See also

- LINQ Query Expressions

- join clause

Compiler Error CS0744

Article • 09/15/2021

Expected contextual keyword 'equals'

The pattern for a `join` clause is `join` ... `in` ... `on` ... `equals`, as shown in this example:

```
C#
```

```
var query = from x in array1
            join y in array2 on x equals y
            select x;
```

To correct this error

1. Add the `equals` keyword to the `join` clause.

Example

The following code generates CS0744:

```
C#
```

```
// cs0744.cs
using System;
using System.Linq;

public class C
{
    public static int Main()
    {
        int[] array1 = { 1, 2, 3 ,4, 5, 6 };
        int[] array2 = { 5, 6, 7, 8, 9 };
        var c = from x in array1
                join y in array2 on x y // CS0744
                select x;
        return 1;
    }
}
```

See also

- LINQ Query Expressions

- join clause

Compiler Error CS0745

Article • 09/15/2021

Expected contextual keyword 'by'

The pattern for the `group` clause is `group...by` followed by an optional `into`, as shown in the following example:

C#

```
string[] names = { "Bob", "Bill", "Jonetta", "Mary" };

var query = from name in names
            group name by name[0];
```

or

C#

```
var query2 = from name in names
             group name by name[0] into g
             //...additional query clauses
```

To correct this error

1. Add the `by` keyword to the `group` clause.

Example

The following code generates CS0745:

C#

```
// cs0745.cs
using System;
using System.Linq;

public class C
{
    public static int Main()
    {
        string[] names = { "Bob", "Bill", "Jonetta", "Mary" };

        var query = from name in names
```

```
                    group name name[0]; // CS0745

        return 1;
    }
}
```

See also

- LINQ Query Expressions
- group clause

Compiler Error CS0746

Article • 09/15/2021

Invalid anonymous type member declarator. Anonymous type members must be declared with a member assignment, simple name or member access.

An anonymous type must be declared with a member assignment, simple name, or member access.

To correct this error

1. Ensure that your declaration uses only member assignment, simple names, or member access expressions.

Example

The following code generates CS0746 in the declaration of `incorrect_1` and `incorrect_2`. The following declarations show two of the correct ways to declare an anonymous type.

C#

```
// cs0746.cs
public class C
{
    public static int Main()
    {
        int i = 100;
        string s = "Bottles of beer.";

        var incorrect_1 = new { a.b = 1 }; // CS0746
        var incorrect_2 = new {100, "Bottles of beer."}; // CS0746
        var correct_1 = new { i, s }; //OK
        var correct_2 = new {num = i, message = s}; // OK

        return 1;
    }
}
```

See also

- Anonymous Types

Compiler Error CS0752

Article • 09/15/2021

A partial method cannot have out parameters

A partial method cannot have an out parameter. Out parameters are not allowed because if the partial method is removed by the compiler then there is no guarantee that the out parameter is ever assigned.

To correct this error

1. Remove the out modifier from the parameter and use the return value of the method instead, or else remove the partial modifier from the method declaration.

Example

The following code generates CS0752:

C#

```
// cs0752.cs
public partial class C
{
    partial void Part(out int num); // CS0752
    // try the following line instead
    // partial void Part(int num);

    public static int Main()
    {
        return 1;
    }
}
```

See also

- Partial Classes and Methods

Compiler Error CS0753

Article • 09/15/2021

Only methods, classes, structs, or interfaces may be partial.

The partial modifier can only be used with classes, structs, interfaces, and methods.

To correct this error

1. Remove the `partial` modifier from the variable or language construct.

Example

The following code generates CS0753:

C#

```
// cs0753.cs
using System;

    public partial class C
    {
        partial int num; // CS0753
        public static int Main()
        {
            return 1;
        }
    }
```

See also

- Partial Classes and Methods

Compiler Error CS0766

Article • 09/15/2021

Partial methods must have a void return type.

A partial method cannot return a value. Its return type must be void.

To correct this error

1. Give the partial method a void return type, or else convert the method to a regular (not partial) method.

Example

The following example generates CS0766:

C#

```
// cs0766.cs
using System;

    public partial class C
    {
        partial int Part(); // CS0766

        // Typically the implementing declaration
        // is contained in a separate file.
        partial int Part() //CS0766
        {
        }

        public static int Main()
        {
            return 1;
        }

    }
```

See also

- Partial Classes and Methods

Compiler Error CS0811

Article • 09/15/2021

The fully qualified name for 'name' is too long for debug information. Compile without '/debug' option.

There are size constraints on variable and type names in debug information.

To correct this error

1. If modifying the name is not possible, the only alternative is to compile without the **DebugType** option.

Example

The following code generates CS0811:

C#

```
// cs0811.cs
//Compile with: /debug
using System;
using System.Collections.Generic;

namespace TestNamespace
{
    using Long =
List<List<List<List<List<List<List<List<List<List<List<List<List<List

<List<List<List<List<List<List<List<List<List<List<List<List<List<List<List<
int>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>>; // CS0811

    class Test
    {
        static int Main()
        {
            return 1;
        }
    }
}
```

Compiler Error CS0815

Article • 11/06/2021

Cannot assign 'expression' to an implicitly typed local

An expression that is used as the initializer for an implicitly typed variable must have a type. Because anonymous function expressions, method group expressions, and the null literal expression do not have a type, they are not appropriate initializers. An implicitly typed variable cannot be initialized with a null value in its declaration, although it can later be assigned a value of null. With C# version 10 Lambda expressions and method groups with natural types can be used as initializers in `var` declarations.

To correct this error

1. Provide an explicit type for the variable.
2. Or specify natural types with C# version 10 and higher.

Example

The following code generates CS0815:

C#

```
// cs0815.cs
class Test
{
    public static int Main()
    {
        var d = s => -1; // CS0815
        var e = (string s) => 0; // CS0815 for C# versions before 10
        var p = null; // CS0815
        var del = delegate(string a) { return -1; }; // CS0815
        return -1;
    }
}
```

See also

- Implicitly Typed Local Variables

Compiler Error CS0818

Article • 09/15/2021

Implicitly typed locals must be initialized

An implicitly typed local variable must be initialized with a value at the same time that it is declared.

To correct this error

1. Assign a value to the variable or else give it an explicit type.

Example

The following code generates CS0818:

```
C#
```

```
// cs0818.cs
class A
{
    public static int Main()
    {
        var a; // CS0818
        return -1;
    }
}
```

See also

- Implicitly Typed Local Variables

Compiler Error CS0819

Article • 10/12/2021

Implicitly-typed variables cannot have multiple declarators.

Multiple declarators are allowed in explicit type declarations, but not with implicitly typed variables.

To correct this error

There are three options:

1. If the variables are of the same type, use explicit declarations.
2. Declare and assign a value to each implicitly typed local variable on a separate line.
3. Declare a variable using Tuple deconstruction syntax. **Note**: this option will not work inside a `using` statement as `Tuple` does not implement `IDisposable`.

Example 1

The following code generates CS0819:

C#

```
// cs0819.cs
class Program
{
    public static void Main()
    {
        var a = 3, b = 2; // CS0819

        // First correction option.
        //int a = 3, b = 2;

        // Second correction option.
        //var a = 3;
        //var b = 2;

        // Third correction option.
        //var (a, b) = (3, 2);
    }
}
```

Example 2

The following code generates CS0819:

C#

```
// cs0819.cs
class Program
{
    public static void Main()
    {
        using (var font1 = new Font("Arial", 10.0f),
            font2 = new Font("Arial", 10.0f)) // CS0819
        {
        }

        // First correction option.
        //using (Font font1 = new Font("Arial", 10.0f),
        //    font2 = new Font("Arial", 10.0f))
        //{
        //}

        // Second correction option.
        //using (var font1 = new Font("Arial", 10.0f)
        //{
        //    using (var font2 = new Font("Arial", 10.0f)
        //    {
        //    }
        //}
    }
}
```

See also

- Implicitly Typed Local Variables
- Deconstructing Tuples and Other Types

Compiler Error CS0822

Article • 09/15/2021

Implicitly typed locals cannot be const

Implicitly typed local variables are only necessary for storing anonymous types. In all other cases they are just a convenience. If the value of the variable never changes, just give it an explicit type. Attempting to use the `readonly` modifier with an implicitly typed local will generate CS0106.

To correct this error

1. If you require the variable to be constant or `readonly`, give it an explicit type.

Example

The following code generates CS0822:

C#

```
// cs0822.cs
class A
{

    public static int Main()
    {
        const var x = 0; // CS0822.cs
        return -1;
    }
}
```

See also

- Implicitly Typed Local Variables

Compiler Error CS0825

Article • 09/15/2021

The contextual keyword 'var' may only appear within a local variable declaration.

To correct this error

1. If the variable belongs at class scope, give it an explicit type. Otherwise move it inside the method where it will be used.

Example

The following code generates CS0825 because `var` is used on a class field:

C#

```
// cs0825.cs
class Test
{
    // Both of these declarations trigger CS0825
    private var genreName;
    private var bookTitles = new List<string>();

    static int Main()
    {
        var totalBooks = 42; // var is OK here
        return -1;
    }
}
```

See also

- Implicitly Typed Local Variables

Compiler Error CS0828

Article • 09/15/2021

Cannot assign 'expression' to anonymous type property.

An anonymous type cannot be initialized with a null value or an unsafe type, or a method group or anonymous function.

To correct this error

1. Either add a type declaration to the left side of the assignment, or change the expression on the right side so that it has an acceptable type.

Example

The following code generates CS0828 because a member of an anonymous type cannot be initialized with a null value.

C#

```
// cs0828.cs
using System;

public class C
{
    public static int Main()
    {
        var a = 1;
        var c = new { p1 = null }; // CS0828
        return 1;
    }
}
```

See also

- Implicitly Typed Local Variables

Compiler Error CS0833

Article • 09/15/2021

An anonymous type cannot have multiple properties with the same name.

An anonymous type, just like any type, cannot have two properties that have the same name.

To correct this error

1. Give each property in the type a unique name.

Example

The following example generates CS0833:

C#

```
// cs0833.cs
using System;

public class C
{
    public static int Main()
    {
        var c = new { p1 = 1, p1 = 2 }; // CS0833
        return 1;
    }
}
```

See also

- Anonymous Types

Compiler Error CS0836

Article • 09/15/2021

Cannot use anonymous type in a constant expression.

The only things allowed in a constant expression are named constants, literals, and mathematical expressions that combine constant expressions.

To correct this error

1. Make the anonymous type a named type.

Example

The following example shows one way to generate CS0836:

C#

```
// cs0836.cs
using System;
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true, Inherited =
false)]
public class A : Attribute
{
    public A(object obj)
    {
    }
}

[A(new { })] // CS0836
public class B
{
}

public class Test
{
    public static int Main()
    {
        return 0;
    }
}
```

Compiler Error CS0837

Article • 05/09/2023

The first operand of an 'is' or 'as' operator may not be a lambda expression, anonymous method, or method group.

Lambda expressions, anonymous methods and method groups may not be used on the left side of is or as.

To correct this error

- If the error involves the `is` operator, remember that `is` takes a value and a type and tells you whether the value can be made into that type by a reference, boxing, or unboxing conversion. Because lambdas are not values and have no reference, boxing, or unboxing conversions, lambdas are not candidates for `is`.
- If the code misuses `as`, the correction is probably to change it to a cast.

Example

The following example generates CS0837:

C#

```
// cs0837.cs
namespace TestNamespace
{
    public delegate void Del();

    class Test
    {
        static int Main()
        {
            bool b1 = (() => { }) is Del;   // CS0837
            bool b2 = delegate() { } is Del;// CS0837
            Del d1 = () => { } as Del;      // CS0837
            Del d2 = delegate() { } as Del; // CS0837
            return 1;
        }
    }
}
```

Compiler Error CS0840

Article • 10/26/2024

'Property name' must declare a body because it is not marked abstract or extern. Automatically implemented properties must define both get and set accessors.

Unless a regular property is marked as `abstract` or `extern`, or is a member of a `partial` type, it must supply a body. Automatically implemented properties do not provide accessor bodies, but they must specify both accessors. To create a read-only automatically implemented property, make the set accessor `private`.

To correct this error

1. Supply the missing body or accessor or else use the abstract, extern, or partial (Type) modifiers on it and/or its enclosing type.

Example

The following example generates CS0840:

C#

```
// cs0840.cs
// Compile with /target:library
using System;
class Test
{
    public int myProp { get; } // CS0840

    // to create a read-only property
    // try the following line instead
    public int myProp2 { get; private set; }

}
```

See also

- Automatically implemented properties

Compiler Error CS0841

Article • 02/04/2022

Cannot use local variable 'name' before it is declared.

A variable must be declared before it is used.

Example of a variable used before declaration

The following example generates CS0841:

C#

```
// cs0841.cs
using System;

public class Program
{
    public static void Main()
    {
        j = 5; // CS0841
        int j;
    }
}
```

Correct the error by moving the declaration before usage

Move the variable declaration before the line where the error occurs.

C#

```
using System;

public class Program
{
    public static void Main()
    {
        int j;
        j = 5;
    }
}
```

Example of a variable shadowing a type

In the following example, the intent was comparing `parameter` with `MyEnum.A`. Because a local variable declared later with the same type name, it shadows the type `MyEnum` and `MyEnum` in this method no longer refers to the `enum`, but refers to the declared local variable.

C#

```
using System;

public enum MyEnum
{
    A, B, C
}

public class C
{
    public void M(MyEnum parameter)
    {
        // error CS0841: Cannot use local variable 'MyEnum' before it is
declared
        if (parameter == MyEnum.A)
        {
            return;
        }

        // Change the variable to 'myEnum' to avoid shadowing the 'MyEnum'
type.
        // This change also aligns with the C# coding conventions.
        var MyEnum = parameter;
        Console.WriteLine(MyEnum.ToString());
    }
}
```

Compiler Error CS0842

Article • 10/26/2024

Automatically implemented properties cannot be used inside a type marked with StructLayout(LayoutKind.Explicit).

Automatically implemented properties have their backing fields provided by the compiler and the field is not accessible to source code. Therefore, they are not compatible with LayoutKind.Explicit.

To correct this error

1. Make the property a regular property in which you provide the accessor bodies.

Example

The following example generates CS0842:

C#

```
// cs0842.cs
using System;
using System.Runtime.InteropServices;

namespace TestNamespace
{
    [StructLayout(LayoutKind.Explicit)]
    struct Str
    {
        public int Num // CS0842
        {
            get;
            set;
        }

        static int Main()
        {
            return 1;
        }
    }
}
```

Compiler Error CS0844

Article • 09/15/2021

Cannot use local variable 'name' before it is declared. The declaration of the local variable hides the field 'name'.

An identifier can have only one meaning in a given block. Local variables that have the same name as class fields can hide the field by introducing a second meaning for the identifier. Therefore the compiler generates an error when you refer to a class field in a method, and then declare a local variable by the same name.

To correct this error

- Use `this.num` to refer to the class field.
- Give the local variable a different name from the class field.

Example

The following code generates CS0844:

C#

```
class Test
    {
        int num;
        public void TestMethod()
        {
            num = 5; // CS0844
            int num = 6;
        }
        public static int Main()
        {
            return 1;
        }
    }
```

Correct the error by using `this.num` to refer to the class field

C#

```
class Test
    {
        int num;
        public void TestMethod()
        {
            this.num = 5; // Error fixed.
            int num = 6;
        }
        public static int Main()
        {
            return 1;
        }
    }
```

Correct the error by giving the local variable a different name from the class field

C#

```
class Test
    {
        int num;
        public void TestMethod()
        {
            num = 5; // Error fixed.
            int num2 = 6;
        }
        public static int Main()
        {
            return 1;
        }
    }
```

Compiler Error CS1001

Article • 04/02/2022

Identifier expected

You did not supply an identifier. An identifier is the name of a class, struct, namespace, method, variable, and so on, that you provide.

The following example declares a simple class but does not give the class a name:

C#

```
public class //CS1001
{
    public int Num { get; set; }
    void MethodA() {}
}
```

The following sample generates CS1001 because, when declaring an enum, you must specify members:

C#

```
public class Program
{
    enum Colors
    {
        'a', 'b' // CS1001, 'a' is not a valid int identifier
        // The following line shows examples of valid identifiers:
        // Blue, Red, Orange
    };

    public static void Main()
    {
    }
}
```

Parameter names are required even if the compiler doesn't use them, for example, in an interface definition. These parameters are required so that programmers who are consuming the interface know something about what the parameters mean.

C#

```
interface IMyTest
{
    void TestFunc1(int, int);  // CS1001
    // Use the following line instead:
```

```
    // void TestFunc1(int a, int b);
}

class CMyTest : IMyTest
{
    void IMyTest.TestFunc1(int a, int b)
    {
    }
}
```

See also

- Operators and expressions
- Types

Compiler Error CS1002

Article • 09/15/2021

; expected

The compiler detected a missing semicolon. A semicolon is required at the end of every statement in C#. A statement may span more than one line.

The following sample generates CS1002:

C#

```
// CS1002.cs
namespace x
{
   abstract public class clx
   {
      int i   // CS1002, missing semicolon

      public static int Main()
      {
         return 0;
      }
   }
}
```

See also

- Statements

Compiler Error CS1003

Article • 09/15/2021

Syntax error, 'char' expected

The compiler will generate this error for any one of several error conditions. Review your code to find the syntax error.

The following sample generates CS1003:

C#

```
// CS1003.cs
public class b
{
   public static void Main()
   {
      int[] a;
      a();   // CS1003
   }
}
```

Compiler Error CS1004

Article • 09/15/2021

Duplicate 'modifier' modifier

A duplicate modifier, such as an **access** modifier, was detected.

The following sample generates CS1004:

C#

```
// CS1004.cs
public class clx
{
   public public static void Main()   // CS1004, two public keywords
   {
   }
}
```

Compiler Error CS1008

Article • 09/15/2021

Type byte, sbyte, short, ushort, int, uint, long, or ulong expected

Certain data types, such as enums, can only be declared to hold data of specified types.

The following sample generates CS1008:

C#

```
// CS1008.cs
abstract public class clx
{
   enum splitch : char   // CS1008, char not valid type for enums
   {
      x, y, z
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1010

Article • 09/15/2021

Newline in constant

A string was not properly delimited.

The following sample generates CS1010:

C#

```
// CS1010.cs
class Sample
{
   static void Main()
   {
      string a = "Hello World;   // CS1010
      // Use the following line, with end quote before semicolon:
      string a = "Hello World"; //
   }
}
```

See also

- Strings (C# Programming Guide)

Compiler Error CS1013

Article • 09/15/2021

Invalid number

The compiler detected a malformed number. For more information on integral types, see the Integral Types Table.

Example

The following sample generates CS1013:

```
C#
```

```
// CS1013.cs
class Sample
{
    static void Main()
    {
        int i = 0x;   // CS1013
    }
}
```

Compiler Error CS1014

Article • 09/15/2021

A get or set accessor expected

A method declaration was found in a property declaration. You can only declare `get` and `set` methods in a property.

For more information on properties, see Using Properties.

Example

The following sample generates CS1014.

C#

```
// CS1014.cs
// compile with: /target:library
class Sample
{
   public int TestProperty
   {
      get
      {
         return 0;
      }
      int z;   // CS1014  not get or set
   }
}
```

Compiler Error CS1015

Article • 04/22/2023

An object, string, or class type expected

An attempt was made to pass a predefined data type into a catch block. Only data types that derive from System.Exception can be passed into a `catch` block. For more information on exceptions, see Exceptions and Exception Handling.

Example

The following sample generates CS1015:

C#

```
// CS1015.cs
class Sample
{
    static void Main()
    {
        try
        {
        }
        catch(int)   // CS1015, int is not derived from System.Exception
        {
        }
    }
}
```

Compiler Error CS1017

Article • 10/27/2021

Catch clauses cannot follow the general catch clause of a try statement

A `catch` block that does not take any parameters must be the last in a series of `catch` blocks. For more information on exceptions, see Exceptions and Exception Handling.

Example

The following sample generates CS1017:

C#

```
// CS1017.cs
using System;

namespace x
{
    public class b : Exception
    {
    }

    public class a
    {
        public static void Main()
        {
            try
            {
            }

            catch   // CS1017, must be last catch
            {
            }

            catch(b)
            {
                throw;
            }
        }
    }
}
```

Compiler Error CS1018

Article • 09/15/2021

Keyword 'this' or 'base' expected

The compiler encountered an incomplete constructor declaration.

Example

The following example generates CS1018, and suggests several ways to resolve the error:

C#

```
// CS1018.cs
public class C
{
}

public class a : C
{
    public a(int i)
    {
    }

    public a () :   // CS1018
    // possible resolutions:
    // public a () resolves by removing the colon
    // public a () : base() calls C's parameterless constructor
    // public a () : this(1) calls the assignment constructor of class a
    {
    }

    public static int Main()
    {
        return 1;
    }
}
```

Compiler Error CS1021

Article • 09/15/2021

Integral constant is too large

A value represented by an integer literal is greater than UInt64.MaxValue.

The following sample generates CS1021:

C#

```
// CS1021.cs
class Program
{
    static void Main(string[] args)
    {
        int a = 18_446_744_073_709_552_000;
    }
}
```

The following code also generates CS1021:

C#

```
using System.Numerics;

class Program
{
    static void Main(string[] args)
    {
        var a = new BigInteger(18_446_744_073_709_552_000);
    }
}
```

For information about how to instantiate a System.Numerics.BigInteger instance whose value exceeds the range of the built-in numeric types, see the Instantiating a BigInteger Object section of the BigInteger reference page.

Compiler Error CS1022

Article • 09/15/2021

Type or namespace definition, or end-of-file expected

A source-code file does not have a matching set of braces.

The following sample generates CS1022:

C#

```
// CS1022.cs
namespace x
{
}
}   // CS1022
```

Compiler Error CS1023

08/05/2025

Embedded statement cannot be a declaration or labeled statement

An embedded statement, such as the statements following an **if** statement, can contain neither declarations nor labeled statements.

To resolve this error, wrap the embedded statement in braces to create a block statement. In C#, unlike C/C++, variable declarations and labeled statements must be contained within a block statement to properly define their scope.

The following sample generates CS1023 twice:

C#

```
// CS1023.cs
public class a
{
   public static void Main()
   {
      if (1)
         int i;      // CS1023, declaration is not valid here

      if (1)
         xx : i++;   // CS1023, labeled statement is not valid here
   }
}
```

Example - Corrected code

To fix this error, use braces to create a block statement:

C#

```
// CS1023 - Fixed.cs
public class a
{
   public static void Main()
   {
      if (1)
      {
         int i;      // Fixed: declaration is now in a block statement
      }

      int j = 0;
      if (1)
```

```
        {
            xx : j++;   // Fixed: labeled statement is now in a block statement
        }
    }
}
```

Compiler Error CS1026

Article • 09/15/2021

) expected

An incomplete statement was found.

A common cause of this error is placing a statement, rather than an expression, within an inline expression in an ASP.NET page. For example, the following is incorrect:

ASP.NET (C#)

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks).Days;%>
```

The following is correct:

ASP.NET (C#)

```
<%=new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1, 1).Ticks).Days %>
```

It is interpreted as follows:

ASP.NET (C#)

```
<% Response.Write(new TimeSpan(DateTime.Now.Ticks - new DateTime(2001, 1,
1).Ticks).Days); %>
```

The following example generates CS1026:

C#

```
// CS1026.cs
#if (a == b   // CS1026, add closing )
#endif

class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1031

07/30/2025

Type expected

A type has not been specified where expected, when overloading an operator.
Missing *type* in this case means that there's no *type* given for the return type of the overloaded operator. This error shouldn't be confused with missing generic type parameter.

Example

The following sample generates CS1031:

C#

```
namespace x
{
    public class I
    {
    }

    public class A
    {
        public static operator +(A a)    // CS1031 - Overloaded operator missing a
type
        {
            return new I();
        }

        public static I operator +(A a)  // Correct - type was specified
        {
            return new I();
        }
    }
}
```

Compiler Error CS1033

Article • 09/15/2021

Source file has exceeded the limit of 16,707,565 lines representable in the PDB; debug information will be incorrect

A source-code file exceeded the maximum allowable number of lines that the compiler can process. To resolve this error, create two or more source-code files from the original file. The maximum number of lines is 268,435,454 lines. If you are using **/debug**, using more than 16,707,566 will result in corrupted debug information.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Error CS1034

Article • 09/15/2021

Compiler limit exceeded: Line cannot exceed 'number' characters

The limit for the number of characters allowed on a line is 16,777,214.

Compiler Error CS1035

Article • 09/15/2021

End-of-file found, '*/' expected

An opening comment delimiter was not matched with a closing delimiter.

The following sample generates CS1035:

C#

```
// CS1035.cs
public class a
{
   public static void Main()
   {
   }
}
/*   // CS1035, needs closing comment
```

Compiler Error CS1036

Article • 09/15/2021

(or . expected

The XML in a **DocumentationFile** comment was badly formed.

Compiler Error CS1041

Article • 09/15/2021

Identifier expected, 'keyword' is a keyword

A reserved word for the C# language was found where an identifier was expected. Replace the keyword with a user-specified identifier.

Example 1

The following sample generates CS1041:

C#

```
// CS1041a.cs
class MyClass
{
    public void f(int long)   // CS1041
    // Try the following instead:
    // public void f(int i)
    {
    }

    public static void Main()
    {
    }
}
```

Example 2

When you are importing from another programming language that does not have the same set of reserved words, you can modify the reserved identifier with the @ prefix, as demonstrated in the following sample.

An identifier with an `@` prefix is called a verbatim identifier.

C#

```
// CS1041b.cs
class MyClass
{
    public void f(int long)   // CS1041
    // Try the following instead:
    // public void f(int @long)
    {
```

```
    }

    public static void Main()
    {
    }
}
```

Compiler Error CS1043

Article • 09/15/2021

{ or ; expected

A property accessor was declared incorrectly. For more information, see Using Properties.

Example

The following sample generates CS1043.

C#

```
// CS1043.cs
// compile with: /target:library
public class MyClass
{
   public int DoSomething
   {
      get return 1;   // CS1043
      set {}
   }

   // OK
   public int DoSomething2
   {
      get { return 1;}
   }
}
```

Compiler Error CS1044

Article • 09/15/2021

Cannot use more than one type in a for, using, fixed, or declaration statement

The compiler found an invalid statement.

The following sample generates CS1044:

C#

```
// CS1044.cs
using System;

public class MyClass : IDisposable
{
   public void Dispose()
   {
      Console.WriteLine("Res1.Dispose()");
   }

   public static void Main()
   {
      using (MyClass mc1 = new MyClass(),
             MyClass mc2 = new MyClass())   // CS1044, remove an
instantiation
      {
      }
   }
}
```

Compiler Error CS1055

Article • 09/15/2021

An add or remove accessor expected

If your event is not declared as a field, it must define both **add** and **remove** accessor functions.

The following sample generates CS1055:

C#

```
// CS1055.cs
delegate void del();
class Test
{
   public event del MyEvent
   {
      int i;   // CS1055
      // uncomment accessors and delete previous line to resolve
      // add
      // {
      //    MyEvent += value;
      // }
      // remove
      // {
      //    MyEvent -= value;
      // }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1056

Article • 09/15/2021

Unexpected character 'character'

The C# compiler encountered an unexpected character, and is unable to identify the token currently being processed. For example, if the compiler encounters a Euro-character in the middle of processing an identifier, it will be unable to classify the identifier, since a Euro-character would only be valid inside a string, and the compiler knows it is not processing a string.

Compiler Error CS1057

Article • 09/15/2021

'member': static classes cannot contain protected members

This error is generated by declaring a protected member inside a static class.

Example

The following example generates CS1057.

C#

```
// CS1057.cs
using System;

static class Class1
{
    protected static int x;   // CS1057
    public static void Main()
    {
    }
}
```

Compiler Error CS1059

Article • 09/15/2021

The operand of an increment or decrement operator must be a variable, property or indexer.

This error is raised when you try to increment or decrement a constant value. It can also occur if you try to increment an expression such as `(a+b)++`.

To correct this error

- Make the variable non-const.
- Remove the increment or decrement operator.
- Store the expression in a variable, and then increment the variable.

Example

The following example generates CS1059 because `i` is a constant, not a variable, and `E` is an `Enum` type, whose elements are also constant values.

C#

```
// CS1059.cs
    class Program
    {
        public enum E : sbyte
        {
            a = 1,
            b = 2
        }

        static void Main(string[] args)
        {
            const int i = 0;
            i++;            // CS1059
            E test = E.a++; // CS1059
        }
    }
```

See also

- Constants

Compiler Error CS1061

08/21/2025

'type' does not contain a definition for 'name' and no accessible extension method 'name' accepting a first argument of type 'type' could be found (are you missing a using directive or an assembly reference?).

This error occurs when you try to call a method or access a class member that does not exist.

Example

The following example generates CS1061 because `Person` does not have a `DisplayName` method. It does have a method that is called `WriteName`. Perhaps that is what the author of this source code meant to write.

C#

```
public class Person
{
    private string _name;

    public Person(string name) => _name = name;

    // Person has one method, called WriteName.
    public void WriteName()
    {
        System.Console.WriteLine(_name);
    }
}

public class Program
{
    public static void Main()
    {
        var p = new Person("PersonName");

        // The following call fails because Person does not have
        // a method called DisplayName.
        p.DisplayName(); // CS1061
    }
}
```

To correct this error

- Make sure you typed the member name correctly.

- If you have access to modify this class, you can add the missing member and implement it.
- If you don't have access to modify this class, you can add an extension method.
- If the member you're trying to access should exist but doesn't, you might be missing a required NuGet package. Search NuGet.org to find packages that contain the missing member.

See also

- The C# type system
- Extension Methods

Compiler Error CS1065

Article • 09/20/2022

Default values are not valid in this context.

Default values are used when declaring optional arguments. Optional arguments are not supported when declaring an anonymous method with the delegate operator. The delegate operator creates an anonymous method.

Example

The following sample generates CS1065:

C#

```
// CS1065.cs (5,27)

class A
{
    delegate void D(int x);
    D d1 = delegate(int x = 42) { };
}
```

To correct this error

Removing the default value corrects this error:

C#

```
class A
{
    delegate void D(int x);
    D d1 = delegate(int x) { };
}
```

Compiler Error CS1107

Article • 07/07/2022

A parameter can only have one 'modifier name' modifier.

It is an error for parameter modifiers such as `this`, `ref`, `in`, and `out` to appear more than one time in a parameter definition.

Example

The following example generates CS1107:

C#

```
// cs1107.cs
public static class Test
{
    // Extension methods.
    public static void TestMethod(this this int t) { } // CS1107

}

public class TestTwo
{
    // Regular Instance Method
    public void TestMethod(ref ref int i) { } // CS1107

    // Regular Instance Method
    public void TestMethod(in in double d) { } // CS1107
}
```

Compiler Error CS1108

Article • 09/15/2021

A parameter cannot have all the specified modifiers; there are too many modifiers on the parameter.

Certain combinations of parameter modifiers, such as `in`, `ref`, and `out`, are not allowed because they have mutually exclusive meanings for the compiler.

Example

The following example generates CS1108:

C#

```
// cs1108.cs
// Compile with: /target:library
public class Test
{
    // Regular Instance Method.
        public void TestMethod(ref out int i) {} // CS1108

}
```

Compiler Error CS1503

Article • 01/29/2025

Argument 'argument' cannot convert from TypeA to TypeB

The type of one argument in a method does not match the type that was passed when the class was instantiated.

The following sample generates CS1503:

C#

```
namespace x
{
  public class a
  {
    public a(char i)
    // try the following constructor instead
    // public a(int i)
    {
    }

    public static void Main()
    {
      a aa = new a(2222);   // CS1503
    }
  }
}
```

Compiler Error CS1503

Article • 01/29/2025

Argument 'argument' cannot convert from TypeA to TypeB

The type of one argument in a method does not match the type that was passed when the class was instantiated.

The following sample generates CS1503:

C#

```
namespace x
{
  public class a
  {
    public a(char i)
    // try the following constructor instead
    // public a(int i)
    {
    }

    public static void Main()
    {
      a aa = new a(2222);   // CS1503
    }
  }
}
```

Compiler Error CS1504

Article • 09/15/2021

Source file 'file' could not be opened ('reason')

A source file could not be opened or read by the compiler. The file may be locked by another application, or there could be some other operating system problem. The message includes the operating system's reason for why the compiler could not open or read the file.

Compiler Error CS1507

08/08/2025

Cannot link resource file 'file' when building a module

LinkResources was used in the same compilation with the **module** option on the **OutputType**, which is not allowed. For example, the following options would generate CS1507:

```
Console
csc /linkresource:rf.resource /target:module in.cs
```

Embedding resources (**Resources**), however, is allowed.

Compiler Error CS1508

Article • 09/15/2021

Resource identifier 'identifier' has already been used in this assembly

In a compilation, the same identifier (***identifier***) was passed to two or more **Resources** or **LinkResources** compiler options.

For example, the following options would generate CS1508:

```
Console
```

```
/resource:anyfile.bmp,DuplicatIdent /linkresource:a.bmp,DuplicatIdent
```

Compiler Error CS1509

Article • 09/15/2021

Referenced file 'file' is not an assembly; use **AddModules** option instead

An output file (output file 1), produced in a compilation that used the **module** element of the **TargetType** (does not have an assembly manifest), was specified to **References**. So, rather than appending an assembly to the assembly for the current program, the metadata information in output file 1 will be added to the assembly for the current program.

Compiler Error CS1511

Article • 09/15/2021

Keyword 'base' is not available in a static method

The base keyword was used in a static method. `base` can only be called in an instance constructor, instance method, or instance accessor.

Example

The following sample generates CS1511.

C#

```
// CS1511.cs
// compile with: /target:library
public class A
{
   public int j = 0;
}

class C : A
{
   public void Method()
   {
      base.j = 3;   // base allowed here
   }

   public static int StaticMethod()
   {
      base.j = 3;   // CS1511
      return 1;
   }
}
```

Compiler Error CS1512

Article • 09/15/2021

Keyword 'base' is not available in the current context

The base keyword was used outside of a method, property, or constructor.

The following example generates CS1512:

C#

```
// CS1512.cs
using System;

class Base {}

class CMyClass : Base
{
    private String xx = base.ToString();   // CS1512
    // Try putting this initialization in the constructor instead:
    // public CMyClass()
    // {
    //    xx = base.ToString();
    // }

    public static void Main()
    {
        CMyClass z = new CMyClass();
    }
}
```

Compiler Error CS1513

Article • 09/15/2021

} expected

The compiler expected a closing curly brace (`}`) that was not found.

The following sample generates CS1513:

```
C#
```

```
// CS1513
namespace y   // CS1513, no close curly brace
{
   class x
   {
      public static void Main()
      {
      }
   }
```

Compiler Error CS1514

Article • 09/15/2021

{ expected

The compiler expected an opening curly brace (`{`) that was not found.

The following sample generates CS1514:

```
C#
```

```
// CS1514.cs
namespace x
// CS1514, no open curly brace
```

Compiler Error CS1515

Article • 09/15/2021

'in' expected

In a foreach, in statement, the "in" part is missing.

Example

The following sample generates CS1515:

C#

```
using System;

class Driver
{
   static void Main()
   {
      int[] arr = new int[] {1, 2, 3};

      // try the following line instead
      // foreach (int x in arr)
      foreach (int x arr)      // CS1515, "in" is missing
      {
         Console.WriteLine(x);
      }
   }
}
```

Compiler Error CS1518

Article • 09/15/2021

Expected class, delegate, enum, interface, or struct

A declaration was found that is not supported in a namespace. Inside a namespace, the compiler accepts only classes, structs, enums, interfaces, namespaces, and delegates.

Example

The following sample generates CS1518:

C#

```
// CS1518.cs
namespace x
{
   sealed class c1 {};      // OK
   namespace f2 {};         // OK
   sealed f3 {};            // CS1518
}
```

Compiler Error CS1519

Article • 09/15/2021

Invalid token 'token' in class, struct, or interface member declaration

This error is generated whenever a token is encountered in a location where it does not belong. A *token* is a keyword; an identifier (the name of a class, struct, method, and so on); a string, character, or numeric literal value such as 108, "Hello", or 'A'; or an operator or punctuator such as `==` or `;`.

Any class, struct, or interface member declaration that contains invalid modifiers before the type will generate this error. To fix the error, remove the invalid modifiers.

The following sample generates CS1519 in five places because tokens are placed in locations where they are not valid:

C#

```
// CS1519.cs
// Generates CS1519 because a class name cannot be a number:
class Test 42
{
// Generates CS1519 because of 'j' following 'I'
// with no comma between them:
    int i j;
// Generates CS1519 because of "checked" on void method:
    checked void f4();

// Generates CS1519 because of "num":
    void f5(int a num){}

// Generates CS1519 because of namespace inside class:
    namespace;

}
```

See also

- Classes
- Structure types
- Interfaces
- Methods

Compiler Error CS1520

Article • 07/06/2022

Method must have a return type

A method that is declared in a class, struct, or interface must have an explicit return type. In the following example, the `IntToString` method has a return value of string:

C#

```
class Test
{
    string IntToString(int i)
    {
        return i.ToString();
    }
}
```

The following sample generates CS1520:

C#

```
public class x
{
   // Method declaration missing a return type before the name of MyMethod
   // Note: the method is empty for the purposes of this example so as to
not add confusion.
   MyMethod() { }
}
```

And can be fixed by adding a return type to the method, such as adding `void` in the example below:

C#

```
public class x
{
   // MyMethod no longer throws an error, because it has a return type --
"void" in this case.
   void MyMethod() { }
}
```

Alternatively, this error might be encountered when the case of a constructor's name differs from that of the class or struct declaration, as in the following sample. Because

the name is not exactly the same as the class name, the compiler interprets it as a regular method, not a constructor, and produces the error:

C#

```
public class Class1
{
   // Constructor should be called Class1, not class1
   public class1()   // CS1520
   {
   }
}
```

See also

- Methods
- Constructors

Compiler Error CS1521

Article • 09/15/2021

Invalid base type

A base class specification was ill formed.

The following sample generates CS1521:

C#

```
// CS1521.cs
class CMyClass
{
   public static void Main()
   {
   }
}

class CMyClass1 : CMyClass     // OK
{
}

class CMyClass2 : CMyClass[]   // CS1521
{
}

class CMyClass3 : CMyClass*    // CS1521
{
}
```

Compiler Error CS1524

Article • 10/27/2021

Expected catch or finally

A `try` block must be followed by a `catch` or `finally` block.

For more information on exceptions, see Exceptions and Exception Handling.

Example

The following sample generates CS1524:

C#

```
// CS1524.cs
class x
{
    public static void Main()
    {
        try
        {
            // Code here
        }
        catch
        {
        }
        try
        {
            // Code here
        }
        finally
        {
        }
        try
        {
            // Code here
        }
    }     // CS1524, missing catch or finally
}
```

Compiler Error CS1525

Article • 01/09/2024

Invalid expression term 'term'

The compiler detected an invalid term in an expression. This error can be caused by a missing expression where one is expected, leading to subsequent tokens being wrongly parsed as an expression, or an invalid construct is used within an expression. Common root causes include unmatched tokens, missing semicolon or excess delimiters.

The following sample generates CS1525:

C#

```
// CS1525.cs
class MyClass
{
    public static void Method(int number) {}

    public static void Main()
    {
        int i = 0;
        i = i + 'c' + 1 + (2);   // OK
        i = i + void + throw;    // CS1525, these keywords are not valid in
this expression

        Method(123,);            // CS1525, excess trailing comma

        goto EmptyLabel;
        EmptyLabel:              // CS1525, empty label
        // Add something here to resolve the error, for example:
        // System.Console.WriteLine("Hello!");
    }
}
```

Compiler Error CS1526

Article • 09/15/2021

A new expression requires (), [], or {} after type

The new operator, used to dynamically allocate memory for an object, was not specified correctly.

Example

The following sample shows how to use `new` to allocate space for an array and an object.

C#

```
// CS1526.cs
public class y
{
    public static int globalCounter = 0;
    public int instanceCounter = 0;
}

public class z
{
    public static void Main()
    {
        y yInstance = new y;   // CS1526
        y[] yArray = new y[10];   // Array of Ys

        for (int i = 0; i < yArray.Length; i++)
            yArray[i] = new y();   // an object of type y
    }
}
```

Compiler Error CS1527

Article • 09/15/2021

Elements defined in a namespace cannot be explicitly declared as private, protected, protected internal or private protected.

Type declarations in a namespace can have either public or internal access. If no accessibility is specified, **internal** is the default.

The following sample generates CS1527:

C#

```
// CS1527.cs
namespace Sample
{
   private class C1 {}             // CS1527
   protected class C2 {}           // CS1527
   protected internal class C3 {}  // CS1527
   private protected class C4 {}   // CS1527
}
```

The following example generates CS1527 because when no namespace is explicitly declared in your program code, all type declarations are located implicitly within the global namespace.

C#

```
//cs1527_2.cs
using System;

protected class C4 {}
private struct S1 {}
```

See also

- Namespaces
- :: Operator
- Accessibility Domain
- Access Modifiers

Compiler Error CS1528

Article • 09/15/2021

Expected ; or = (cannot specify constructor arguments in declaration)

A reference to a class was formed as if an object to the class was being created. For example, there was an attempt to pass a variable to a constructor. Use the new operator to create an object of a class.

The following sample generates CS1528:

C#

```
// CS1528.cs
using System;

public class B
{
   public B(int i)
   {
      _i = i;
   }

   public void PrintB()
   {
      Console.WriteLine(_i);
   }

   private int _i;
}

public class mine
{
   public static void Main()
   {
      B b(3);   // CS1528, reference is not an object
      // try one of the following
      // B b;
      // or
      // B bb = new B(3);
      // bb.PrintB();
   }
}
```

Compiler Error CS1530

Article • 09/15/2021

Keyword 'new' is not allowed on elements defined in a namespace

It is not necessary to specify the new keyword on any construct that is in a namespace.

The following sample generates CS1530:

C#

```
// CS1530.cs
namespace a
{
   new class i   // CS1530
   {
   }

   // try the following instead
   class ii
   {
      public static void Main()
      {
      }
   }
}
```

Compiler Error CS1536

Article • 09/15/2021

Invalid parameter type void

It is not necessary or valid to specify a void parameter other than a void pointer.

The following sample generates CS1536:

C#

```
// CS1536.cs
class a
{
   public static int x( void )   // CS1536
   // try the following line instead
   // public static int x()
   {
      return 0;
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1540

Article • 09/15/2021

Cannot access protected member 'member' via a qualifier of type 'type1'; the qualifier must be of type 'type2' (or derived from it)

A derived class cannot access protected members of its base class through an instance of the base class. An instance of the base class declared in the derived class might, at run time, be an instance of another type that is derived from the same base but is not otherwise related to the derived class. Because protected members can be accessed only by derived types, any attempts to access protected members that might not be valid at run time are marked by the compiler as not valid.

In the `Employee` class in the following example, `emp2` and `emp3` both have type `Person` at compile time, but `emp2` has type `Manager` at run time. Because `Employee` is not derived from `Manager`, it cannot access the protected members of the base class, `Person`, through an instance of the `Manager` class. The compiler cannot determine what the run-time type of the two `Person` objects will be. Therefore, both the call from `emp2` and the call from `emp3` cause compiler error CS1540.

C#

```
using System;
using System.Text;

namespace CS1540
{
    class Program1
    {
        static void Main()
        {
            Employee.PreparePayroll();
        }
    }

    class Person
    {
        protected virtual void CalculatePay()
        {
            Console.WriteLine("CalculatePay in Person class.");
        }
    }

    class Manager : Person
    {
        protected override void CalculatePay()
        {
```

```
            Console.WriteLine("CalculatePay in Manager class.");

        }
    }

    class Employee : Person
    {
        public static void PreparePayroll()
        {
            Employee emp1 = new Employee();
            Person emp2 = new Manager();
            Person emp3 = new Employee();
            // The following line calls the method in the Employee base
class,
            // Person.
            emp1.CalculatePay();

            // The following lines cause compiler error CS1540. The compiler
            // cannot determine at compile time what the run-time types of
            // emp2 and emp3 will be.
            //emp2.CalculatePay();
            //emp3.CalculatePay();

        }
    }
}
```

See also

- Inheritance
- Polymorphism
- Access Modifiers
- Abstract and Sealed Classes and Class Members

Compiler Error CS1541

Article • 09/15/2021

Invalid reference option: 'symbol' — cannot reference directories

The compiler detected an attempt to specify a directory rather than a specific file. For example, when you use the **References** compiler option, you must specify a file; it is not possible to specify a directory.

For example, passing `/reference:c:\` to the compiler would generate CS1541.

Compiler Error CS1542

Article • 09/15/2021

'dll' cannot be added to this assembly because it already is an assembly; use '/R' option instead

The file that was referenced with the **AddModules** compiler option was not built with **module** element of the **TargetType** compiler option; use **References** to reference the file in this compilation.

Compiler Error CS1545

Article • 09/15/2021

Property, indexer, or event 'property' is not supported by the language; try directly calling accessor methods 'set accessor' or 'get accessor'

The code is consuming an object that has a non-default indexer and tried to use the indexed syntax. To resolve this error, call the property's `get` or `set` accessor method.

Example 1

C++

```
// CPP1545.cpp
// compile with: /clr /LD
// a Visual C++ program
using namespace System;
public ref struct Employee {
   Employee( String^ s, int d ) {}

   property String^ name {
      String^ get() {
         return nullptr;
      }
   }
};

public ref struct Manager {
   property Employee^ Report [String^] {
      Employee^ get(String^ s) {
         return nullptr;
      }

      void set(String^ s, Employee^ e) {}
   }
};
```

Example 2

The following sample generates CS1545.

C#

```
// CS1545.cs
// compile with: /r:CPP1545.dll
```

```
class x {
   public static void Main() {
      Manager Ed = new Manager();
      Employee Bob = new Employee("Bob Smith", 12);
      Ed.Report[ Bob.name ] = Bob;   // CS1545
      Ed.set_Report( Bob.name, Bob);   // OK
   }
}
```

Compiler Error CS1546

Article • 09/15/2021

Property, indexer, or event 'property' is not supported by the language; try directly calling accessor method 'accessor'

Your code is consuming an object that has a default indexed property and tried to use the indexed syntax. To resolve this error, call the property's accessor method. For more information on indexers and properties, see Indexers.

The following sample generates CS1546.

Example 1

This code sample consists of a .cpp file, which compiles to a .dll, and a .cs file, which uses that .dll. The following code is for the .dll file and defines a property to be accessed by the code in the .cs file.

C++

```
// CPP1546.cpp
// compile with: /clr /LD
using namespace System;
public ref class MCPP
{
public:
    property int Prop [int,int]
    {
        int get( int i, int b )
        {
            return i;
        }
    }
};
```

Example 2

This is the C# file.

C#

```
// CS1546.cs
// compile with: /r:CPP1546.dll
using System;
```

```
public class Test
{
    public static void Main()
    {
        int i = 0;
        MCPP mcpp = new MCPP();
        i = mcpp.Prop(1,1); // CS1546
        // Try the following line instead:
        // i = mcpp.get_Prop(1,1);
    }
}
```

Compiler Error CS1547

Article • 09/15/2021

Keyword 'void' cannot be used in this context

The compiler detected an invalid use of the void keyword.

The following sample generates CS1547:

C#

```
// CS1547.cs
public class MyClass
{
   void BadMethod()
   {
      void i;   // CS1547, cannot have variables of type void
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1548

Article • 09/15/2021

Cryptographic failure while signing assembly 'assembly' — 'reason'

CS1548 occurs when assembly signing fails. This is usually due to an invalid key file name, an invalid key file path, or a corrupt key file.

To fully sign an assembly, you must provide a valid key file that contains information about the public and private keys. To delay sign an assembly, you must select the **Delay sign only** check box and provide a valid key file that contains information about the public key information. The private key is not necessary when an assembly is delay-signed.

For more information, see How to: Sign an Assembly with a Strong Name, **KeyFile** (C# Compiler Options) and **DelaySign** (C# Compiler Options).

When creating an assembly, the C# compiler calls into a utility called al.exe. If there is a failure in the assembly creation, the reason for the failure is reported by al.exe. See Al.exe Tool Errors and Warnings and search that topic for the text reported by the compiler in 'reason'.

See also

- How to: Sign an Assembly with a Strong Name

Compiler Error CS1551

Article • 09/15/2021

Indexers must have at least one parameter

An indexer that takes no arguments was declared.

The following sample generates CS1551:

C#

```
// CS1551.cs
public class MyClass
{
   int intI;

   int this[]   // CS1551
   // try the following line instead
   // int this[int i]
   {
      get
      {
         return intI;
      }
      set
      {
         intI = value;
      }
   }

   public static void Main()
   {
   }
}
```

Compiler Error CS1555

08/08/2025

Could not find 'class' specified for Main method

A class was specified to the StartupObject compiler option, but the class name was not found in the source code.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

Compiler Error CS1556

08/08/2025

'construct' specified for Main method must be a valid class or struct

The StartupObject compiler option was passed an identifier that was not a class name.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

Compiler Error CS1557

08/08/2025

Cannot use 'class' for Main method because it is in a different output file

The **StartupObject** compiler option was specified for one output file in a multi-output file compilation. However, the class was not found in the source code for the /main compilation; it was found in a source code file for one of the other output files in the compilation.

Compiler Error CS1558

08/08/2025

'class' does not have a suitable static Main method

The **StartupObject** compiler option specified a class in which to look for a **Main** method. However, the Main method was not defined correctly.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

The following example generates CS1558 because of invalid return type.

C#

```
// CS1558.cs
// compile with: /main:MyNamespace.MyClass

namespace MyNamespace
{
   public class MyClass
   {
      public static float Main()
      {
         return 0.0; // CS1558 because the return type is a float.
      }
   }
}
```

Compiler Error CS1559

Article • 09/15/2021

Cannot use 'object' for Main method because it is imported

An invalid class was specified to the **StartupObject** compiler option; the class cannot be used as a location for the Main method.

Compiler Error CS1561

Article • 09/15/2021

Output filename is too long or invalid

The output file name cannot be longer than 256 characters, and may not contain invalid characters, such as the Euro symbol, a question mark, or a backslash, among others.

Compiler Error CS1562

Article • 09/15/2021

Outputs without source must have the /out option specified

The compilation could create an output file, but there was no source code file as input from which the name of the output file could be implied. For example, you may be trying to compile a metadata- or resource-only file.

Use the **OutputAssembly** compiler option to specify the name of the output file.

Compiler Error CS1563

Article • 09/15/2021

Output 'output file' does not have any source files

The **OutputAssembly** compiler option was specified, but no source code files follow. You should either not use **OutputAssembly**, or you should specify the source code files following **OutputAssembly**.

Compiler Error CS1564

Article • 09/15/2021

Conflicting options specified: Win32 resource file; Win32 manifest.

If you use the **/Win32res** compiler option, you must include the custom Win32 manifest (if it is required) in the resource file. You cannot provide a custom Win32 manifest separately from a Win32 resource file. Use the **/win32manifest** option only if you are not specifying a win32 resource file.

To correct this error

1. Add the win32 manifest to the win32 resource file and remove the **/win32manifest** compiler option.

Example

The following example produces CS1564 if it is compiled with the **/Win32res** option and no manifest is included in the resource file.

C#

```
// cs1564.cs
// Compile with: /Win32res
public class Test
{
    static int Main(string[] args)
    {
        return 1;
    }
}
```

The C# 3.0 compiler adds a default win32Manifest to all binaries.

See also

- **Win32Manifest** (C# Compiler Options)
- **Win32Resource** (C# Compiler Options)

Compiler Error CS1565

Article • 09/15/2021

Conflicting options specified: Win32 resource file; Win32 icon

It is not valid to specify both the **Win32Resource** compiler option and the **Win32Icon** compiler option in the same compilation.

Compiler Error CS1566

Article • 09/15/2021

Error reading resource file 'file' — 'reason'

The compiler had trouble with the file name passed to the **Resources** compiler option.

Compiler Error CS1567

Article • 09/15/2021

Error generating Win32 resource: 'file'

Your compilation either used the **Win32Icon** compiler option or did not use **Win32Resource**, which causes the compiler to generate a file that contains resource information, but the compiler was unable to create the file due to insufficient disk space or some other error.

If you are unable to resolve the file-generation problem, you could use **Win32Resource**, which does not generate a file that contains resource information.

Compiler Error CS1569

Article • 09/15/2021

Error generating XML documentation file 'Filename' ('reason')

When attempting to write the XML documentation to the file named in the message, an error occurred for the reason specified. The reason may be something like "network drive not found," or "access denied." Often, the reason will suggest what needs to be done to correct the error. For example, if the error says "access denied," you would verify that you have write permission on the file.

Example 1

C#

```
// 1569a.cs
// compile with: /doc:CS1569.xml
// post-build command: attrib +r CS1569.xml
class Test
{
   /// <summary>Test.</summary>
   public static void Main() {}
}
```

Example 2

The previous sample generated an .xml file that was then set to read only. This sample attempts to write to the same file. The following sample generates CS1569.

C#

```
// CS1569.cs
// compile with: /doc:CS1569.xml
// CS1569 expected
class Test
{
   /// <summary>Test.</summary>
   public static void Main() {}
}
```

Compiler Error CS1575

Article • 09/15/2021

A stackalloc expression requires [] after type

The size of the requested allocation, with stackalloc, must be specified in square brackets.

The following sample generates CS1575:

C#

```
// CS1575.cs
// compile with: /unsafe
public class MyClass
{
   unsafe public static void Main()
   {
      int *p = stackalloc int (30);   // CS1575
      // try the following line instead
      // int *p = stackalloc int [30];
   }
}
```

Compiler Error CS1577

Article • 09/15/2021

Assembly generation failed —reason

The assembly-generation part of the compilation failed after the C# compiler successfully compiled the source code. See the errors documentation for the ALink utility (Al.exe Tool Errors and Warnings) for more information.

See also

- Al.exe (Assembly Linker)

Compiler Error CS1579

Article • 09/15/2021

foreach statement cannot operate on variables of type 'type1' because 'type2' does not contain a public definition for 'identifier'

To iterate through a collection using the foreach statement, the collection must meet the following requirements:

- Its type must include a public parameterless `GetEnumerator` method whose return type is either class, struct, or interface type.
- The return type of the `GetEnumerator` method must contain a public property named `Current` and a public parameterless method named `MoveNext` whose return type is Boolean.

Example

The following sample generates CS1579 because the `MyCollection` class doesn't contain the public `GetEnumerator` method:

C#

```
// CS1579.cs
using System;
public class MyCollection
{
   int[] items;
   public MyCollection()
   {
      items = new int[5] {12, 44, 33, 2, 50};
   }

   // Delete the following line to resolve.
   MyEnumerator GetEnumerator()

   // Uncomment the following line to resolve:
   // public MyEnumerator GetEnumerator()
   {
      return new MyEnumerator(this);
   }

   // Declare the enumerator class:
   public class MyEnumerator
   {
      int nIndex;
      MyCollection collection;
```

```
        public MyEnumerator(MyCollection coll)
        {
            collection = coll;
            nIndex = -1;
        }

        public bool MoveNext()
        {
            nIndex++;
            return (nIndex < collection.items.Length);
        }

        public int Current => collection.items[nIndex];
    }

    public static void Main()
    {
        MyCollection col = new MyCollection();
        Console.WriteLine("Values in the collection are:");
        foreach (int i in col)   // CS1579
        {
            Console.WriteLine(i);
        }
    }
}
```

Compiler Error CS1583

Article • 09/15/2021

'file' is not a valid Win32 resource file

This error occurs when you specify a filename with the **Win32Resource** compiler option that is not a valid or correctly formatted Win32 resource file. In Visual Studio, the filename is specified in the Application pane of the Project Designer.

See also

- Application Page, Project Designer (C#)

Compiler Error CS1585

Article • 09/15/2021

Member modifier 'keyword' must precede the member type and name

The access specifier in a method signature did not occur in the correct location.

The following sample generates CS1585:

C#

```
// CS1585.cs
public class Class1
{
   public void static Main(string[] args)   // CS1585
   // try the following line instead
   // public static void Main(string[] args)
   {
   }
}
```

Compiler Error CS1588

Article • 09/15/2021

Cannot determine common language runtime directory -- 'reason'

The compiler cannot retrieve the directory that the .NET runtime is installed in. This error indicates that the common language runtime is improperly installed.

Compiler Error CS1593

Article • 09/15/2021

Delegate 'del' does not take 'number' arguments

The number of arguments passed to a delegate invocation does not agree with the number of parameters in the delegate declaration.

The following sample generates CS1593:

C#

```
// CS1593.cs
using System;
delegate string func(int i);   // declare delegate

class a
{
   public static void Main()
   {
      func dt = new func(z);
      x(dt);
   }

   public static string z(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func hello)
   {
      hello(8, 9);   // CS1593
      // try the following line instead
      // hello(8);
   }
}
```

Compiler Error CS1594

Article • 09/15/2021

Delegate 'delegate' has some invalid arguments

The type of an argument passed to a delegate invocation does not agree with the type of the parameter in the delegate declaration.

The following sample generates CS1594:

C#

```
// CS1594.cs
using System;
delegate string func(int i);   // declare delegate

class a
{
   public static void Main()
   {
      func dt = new func(z);
      x(dt);
   }

   public static string z(int j)
   {
      Console.WriteLine(j);
      return j.ToString();
   }

   public static void x(func hello)
   {
      hello("8");   // CS1594
      // try the following line instead
      // hello(8);
   }
}
```

Compiler Error CS1597

Article • 02/29/2024

Semicolon after method or accessor block is not valid

Semicolons are not needed (or allowed) to end a method or accessor block.

The following sample generates CS1597:

C#

```
// CS1597.cs
class MyClass
{
    public int Prop { get; };   // CS1597

    public static void Main() { };   // CS1597
}
```

To resolve this error, remove the extra trailing semicolon.

Compiler Error CS1599

Article • 09/15/2021

Method or delegate cannot return type 'type'

Some types in the .NET class library, for example, TypedReference, RuntimeArgumentHandle and ArgIterator cannot be used as return types because they can potentially be used to perform unsafe operations.

The following sample generates CS1599:

C#

```
// CS1599.cs
using System;

class MyClass
{
   public static void Main()
   {
   }

   public TypedReference Test1()   // CS1599
   {
      return null;
   }

   public ArgIterator Test2()   // CS1599
   {
      return null;
   }
}
```

Compiler Error CS1600

Article • 09/15/2021

Compilation cancelled by user

A compilation with the C# compiler was cancelled while using the Visual Studio IDE.

Compiler Error CS1601

Article • 09/15/2021

Method or delegate parameter cannot be of type 'type'

Some types in the .NET class library, for example, TypedReference, RuntimeArgumentHandle and ArgIterator cannot be used as in, ref or out parameters because they could potentially be used to perform unsafe operations.

The following sample generates CS1601:

C#

```
// CS1601.cs
using System;

class MyClass
{
    public void Test1(in TypedReference t)   // CS1601
    {
    }

    public void Test2(ref TypedReference t)   // CS1601
    {
    }

    public void Test3(out ArgIterator t)   // CS1601
    {
        t = default;
    }
}
```

Compiler Error CS1604

Article • 09/15/2021

Cannot assign to 'variable' because it is read-only

An assignment was made to a read-only variable. To avoid this error, do not assign or attempt to modify to this variable in this context.

Compiler Error CS1606

Article • 09/15/2021

Assembly signing failed; output may not be signed -- reason

The assembly was produced, but when the compiler attempted to finish signing it, there was a failure.

Compiler Error CS1608

Article • 09/15/2021

The Required attribute is not permitted on C# types

RequiredAttributeAttribute is not allowed on types defined in C# code.

Compiler Error CS1609

Article • 09/15/2021

Modifiers cannot be placed on event accessor declarations

Modifiers can only be placed on event declarations and not on the event accessor declarations. For more information, see Using Properties.

Example

The following sample generates CS1609.

C#

```
// CS1609.cs
// compile with: /target:library
delegate int Del();
class A
{
   public event Del MyEvent
   {
      private add {}   // CS1609
      // try the following line instead
      // add {}
      remove {}
   }
}
```

Compiler Error CS1612

Article • 03/30/2023

Cannot modify the return value of 'expression' because it is not a variable

An attempt was made to modify a value type that is produced as the result of an intermediate expression but is not stored in a variable. This error can occur when you attempt to directly modify a struct in a generic collection, as shown in the following example:

C#

```
List<MyStruct> list = {…};
list[0].Name = "MyStruct42"; //CS1612
```

To modify the struct, first assign it to a local variable, modify the variable, then assign the variable back to the item in the collection.

C#

```
List<MyStruct> list = {…};
MyStruct ms = list[0];
ms.Name = "MyStruct42";
list[0] = ms;
```

This error occurs because value types are copied on assignment. When you retrieve a value type from a property or indexer, you are getting a copy of the object, not a reference to the object itself. The copy that is returned is not stored by the property or indexer because they are actually methods, not storage locations (variables). You must store the copy into a variable that you declare before you can modify it.

The error does not occur with reference types because a property or indexer in that case returns a reference to an existing object, which is a storage location.

If you are defining the class or struct, you can resolve this error by modifying your property declaration to provide access to the members of a struct. If you are writing client code, you can resolve the error by creating your own instance of the struct, modifying its fields, and then assigning the entire struct back to the property. As a third alternative, you can change your struct to a class.

Example

CS1612 also occurs when you attempt to access the member of a struct through a property on an enclosing class that is returning the entire struct, as shown in the following example:

C#

```
// CS1612.cs
using System;

public struct MyStruct
{
    public int Width;
}

public class ListView
{
    public MyStruct Size { get; set; }
}

public class MyClass
{
    public MyClass()
    {
        ListView lvi;
        lvi = new ListView();
        lvi.Size.Width = 5; // CS1612

        // You can use the following lines instead.
        // MyStruct ms;
        // ms.Width = 5;
        // lvi.Size = ms;
    }

    public static void Main()
    {
        MyClass mc = new MyClass();
        // Keep the console open in debug mode.
        Console.WriteLine("Press any key to exit.");
        Console.ReadKey();
    }
}
```

See also

- Structure types
- Value types
- Reference types

Compiler Error CS1613

Article • 09/15/2021

The managed coclass wrapper class 'class' for interface 'interface' cannot be found (are you missing an assembly reference?)

An attempt was made to instantiate a COM object from an interface. The interface has the **ComImport** and `CoClass` attributes, but the compiler cannot find the type given for the `CoClass` attribute.

To resolve this error, you can try one of the following:

- Add a reference to the assembly that has the coclass (most of the time the interface and coclass should be in the same assembly). See **References** or Add Reference Dialog Box for information.
- Fix the `CoClass` attribute on the interface.

The following sample demonstrates correct usage of **CoClassAttribute**:

C#

```
// CS1613.cs
using System;
using System.Runtime.InteropServices;

[Guid("1FFD7840-E82D-4268-875C-80A160C23296")]
[ComImport()]
[CoClass(typeof(A))]
public interface IA{}
public class A : IA {}

public class AA
{
   public static void Main()
   {
      IA i;
      i = new IA(); // This is equivalent to new A().
                    // because of the CoClass attribute on IA
   }
}
```

Compiler Error CS1614

Article • 09/15/2021

'name' is ambiguous between 'name' and 'nameAttribute'; use either '@name' or 'nameAttribute'.

The compiler has encountered an ambiguous attribute specification.

For convenience, the C# compiler allows you to specify **ExampleAttribute** as just `[Example]`. However, ambiguity arises if an attribute class named `Example` exists along with **ExampleAttribute**, because the compiler cannot tell if `[Example]` refers to the `Example` attribute or the **ExampleAttribute** attribute. To clarify, use `[@Example]` for the `Example` attribute and `[ExampleAttribute]` for **ExampleAttribute**.

The following sample generates CS1614:

C#

```
// CS1614.cs
using System;

// Both of the following classes are valid attributes with valid
// names (MySpecial and MySpecialAttribute). However, because the lookup
// rules for attributes involves auto-appending the 'Attribute' suffix
// to the identifier, these two attributes become ambiguous; that is,
// if you specify MySpecial, the compiler can't tell if you want
// MySpecial or MySpecialAttribute.

public class MySpecial : Attribute {
   public MySpecial() {}
}

public class MySpecialAttribute : Attribute {
   public MySpecialAttribute() {}
}

class MakeAWarning {
   [MySpecial()] // CS1614
                 // Ambiguous: MySpecial or MySpecialAttribute?
   public static void Main() {
   }

   [@MySpecial()] // This isn't ambiguous, it binds to the first attribute
above.
   public static void NoWarning() {
   }

   [MySpecialAttribute()] // This isn't ambiguous, it binds to the second
attribute above.
```

```
    public static void NoWarning2() {
    }

    [@MySpecialAttribute()] // This is also legal.
    public static void NoWarning3() {
    }
}
```

Compiler Error CS1615

Article • 09/15/2021

Argument 'number' should not be passed with the 'keyword' keyword

One of the keywords `ref` or **out** was used when the function did not take a `ref` or **out** parameter for that argument. To resolve this error, remove the incorrect keyword and use the appropriate keyword that matches the function declaration, if any.

The following sample generates CS1615:

C#

```
// CS1615.cs
class C
{
   public void f(int i) {}
   public static void Main()
   {
      int i = 1;
      f(ref i);  // CS1615
   }
}
```

Compiler Error CS1617

07/18/2025

Invalid option 'option' for **LangVersion**. Use `?` to list supported values.

This error occurs if you used the **LangVersion** command line switch or project setting but didn't specify a valid language option. To resolve this error, check the command line syntax or project setting and change it to one of the listed options.

For example, compiling with `csc -langversion:ISO` will generate error CS1617.

How to list supported language versions

To see a list of supported language versions, you reference the table in this article, compile with `-langversion:?`, or temporarily set `<LangVersion>?</LangVersion>` in your project file before building.

Use the reference table (recommended)

The most reliable way to see supported language versions is to consult the reference table at the end of this article, which lists all currently supported language versions.

Use the C# compiler directly

Use the `-langversion:?` option with the C# compiler. You need to find the path to `csc.dll` in your .NET SDK installation:

Console

```
dotnet exec "/path/to/dotnet/sdk/version/Roslyn/bincore/csc.dll" -langversion:?
```

For example, on Linux with .NET 8 SDK:

Console

```
dotnet exec "/usr/lib/dotnet/sdk/8.0.117/Roslyn/bincore/csc.dll" -langversion:?
```

The exact path varies based on your operating system and .NET SDK version.

Use a project file with diagnostic output (not recommended)

You can temporarily set `<LangVersion>?</LangVersion>` in your project file and build with diagnostic verbosity:

```
Console

dotnet build -v diagnostic
```

Look for the "Supported language versions:" line in the output.

> **⚠ Warning**
>
> Setting `<LangVersion>?</LangVersion>` in a project file will cause the build to fail after displaying the supported versions. This is because the compiler exits after listing the versions instead of continuing to compile your code. Remove this setting after viewing the list.

Valid values for -langversion

The valid values for the language versions depend on the .NET version you are using. See the language version rules for more information on which language version is available with which version of .NET. If you are receiving this error while attempting to use a newer language version, either downgrade to a lower language version or update your .NET SDK to a version that supports the language version.

The following table specifies the current valid values for `-langversion`:

Expand table

Value	Meaning
`preview`	The compiler accepts all valid language syntax from the latest preview version.
`latest`	The compiler accepts syntax from the latest released version of the compiler (including minor version).
`latestMajor` or `default`	The compiler accepts syntax from the latest released major version of the compiler.
`14.0`	The compiler accepts only syntax that is included in C# 14 or lower.
`13.0`	The compiler accepts only syntax that is included in C# 13 or lower.
`12.0`	The compiler accepts only syntax that is included in C# 12 or lower.

Value	Meaning
`11.0`	The compiler accepts only syntax that is included in C# 11 or lower.
`10.0`	The compiler accepts only syntax that is included in C# 10 or lower.
`9.0`	The compiler accepts only syntax that is included in C# 9 or lower.
`8.0`	The compiler accepts only syntax that is included in C# 8.0 or lower.
`7.3`	The compiler accepts only syntax that is included in C# 7.3 or lower.
`7.2`	The compiler accepts only syntax that is included in C# 7.2 or lower.
`7.1`	The compiler accepts only syntax that is included in C# 7.1 or lower.
`7`	The compiler accepts only syntax that is included in C# 7.0 or lower.
`6`	The compiler accepts only syntax that is included in C# 6.0 or lower.
`5`	The compiler accepts only syntax that is included in C# 5.0 or lower.
`4`	The compiler accepts only syntax that is included in C# 4.0 or lower.
`3`	The compiler accepts only syntax that is included in C# 3.0 or lower.
`ISO-2` or `2`	The compiler accepts only syntax that is included in ISO/IEC 23270:2006 C# (2.0).
`ISO-1` or `1`	The compiler accepts only syntax that is included in ISO/IEC 23270:2003 C# (1.0/1.2).

Compiler Error CS1618

Article • 09/15/2021

Cannot create delegate with 'method' because it has a Conditional attribute

You cannot create a delegate with a conditional method because the method might not exist in some builds.

The following sample generates CS1618:

C#

```
// CS1618.cs
using System;
using System.Diagnostics;

delegate void del();

class MakeAnError {
   public static void Main() {
      del d = new del(ConditionalMethod);   // CS1618
      // Invalid because on builds where DEBUG is not set,
      // there will be no "ConditionalMethod".
   }
   // To fix the error, remove the next line:
   [Conditional("DEBUG")]
   public static void ConditionalMethod()
   {
      Console.WriteLine("Do something only in debug");
   }
}
```

Compiler Error CS1619

Article • 09/15/2021

Cannot create temporary file 'filename' -- reason

The compiler was unable to create a temporary file for the given reason (for example, the disk is full).

Compiler Error CS1620

Article • 09/15/2021

Argument 'number' must be passed with the 'keyword' keyword

This error occurs if you are passing an argument to a function that takes a ref or out parameter and you don't include the `ref` or `out` keyword at the point of call, or you include the wrong keyword. The error text indicates the appropriate keyword to use and which argument caused the failure.

The following sample generates CS1620:

C#

```
// CS1620.cs
class C
{
    void f(ref int i) {}
    public static void Main()
    {
        int x = 1;
        f(out x);  // CS1620 – f takes a ref parameter, not an out parameter
        // Try this line instead:
        // f(ref x);
    }
}
```

Compiler Error CS1630

Article • 09/15/2021

Invalid option 'option' for **ErrorReport**; must be **prompt**, **send**, **queue**, or **none**

The command line option **ErrorReport** must be followed by **prompt**, **send**, **queue**, or **none**, specifying what action you want taken when an internal compiler error occurs.

> {!NOTE} This compiler option is no longer supported.

Errors and warnings related to the `yield return` statement and iterator methods

Article • 12/18/2024

There are numerous *errors* related to the `yield return` statement and iterator methods:

- CS1622: *Cannot return a value from an iterator. Use the yield return statement to return a value, or yield break to end the iteration.*
- CS1624: *The body of 'accessor' cannot be an iterator block because 'type' is not an iterator interface type*
- CS1625: *Cannot yield in the body of a finally clause*
- CS1626: *Cannot yield a value in the body of a try block with a catch clause*
- CS1627: *Expression expected after yield return*
- CS1629: *Unsafe code may not appear in iterators*
- CS1631: *Cannot yield a value in the body of a catch clause*
- CS1637: *Iterators cannot have unsafe parameters or yield types*
- CS4013: *Instance of type cannot be used inside a nested function, query expression, iterator block or async method*
- CS8154: *The body cannot be an iterator block because it returns by reference*
- CS8176: *Iterators cannot have by-reference locals*
- CS9238: *Cannot use 'yield return' in an 'unsafe' block*
- CS9239: *The `&` operator cannot be used on parameters or local variables in iterator methods.*

Structure of an iterator method

An iterator method must conform to several rules in C#. The compiler issues the following errors when your iterator method violates one or more of those rules:

- **CS1622**: *Cannot return a value from an iterator. Use the yield return statement to return a value, or yield break to end the iteration.*
- **CS1624**: *The body of 'accessor' cannot be an iterator block because 'type' is not an iterator interface type*
- **CS1627**: *Expression expected after yield return*
- **CS1637**: *Iterators cannot have unsafe parameters or yield types*
- **CS8154**: *The body cannot be an iterator block because it returns by reference*

Your iterator method must follow the following rules:

- An iterator method (using `yield return` and optionally `yield break`) can't also use a `return` statement to return a sequence.
- An iterator method must declare an *iterator interface type* as the return type. The *iterator interface types* are: IEnumerable, IEnumerable<T>, IEnumerator, IEnumerator<T>.
- A `yield return` statement must include an expression to return as part of a sequence. `yield return;` isn't valid.
- An iterator method can't use unsafe types as parameters, such as pointers.
- An iterator method can't `yield return` unsafe type, such as pointers.
- An iterator method can't `yield return` by `ref`. You must return by value.

Restrictions on iterator methods

The body of an iterator method must conform to restrictions on the `yield return` statement and its context. The compiler issues the following errors when your iterator violates one of these restrictions:

- **CS1625**: *Cannot yield in the body of a finally clause*
- **CS1626**: *Cannot yield a value in the body of a try block with a catch clause*
- **CS1631**: *Cannot yield a value in the body of a catch clause*
- **CS1629**: *Unsafe code may not appear in iterators*
- **CS9238**: *Cannot use 'yield return' in an 'unsafe' block*
- **CS9239**: *The `&` operator cannot be used on parameters or local variables in iterator methods.*

These errors indicate that your code violates safety rules because an iterator returns an element and resumes to generate the next element:

- You can't `yield return` from a `catch` or `finally` clause.
- You can't `yield return` from a `try` block with a catch clause.
- You can't `yield return` from an `unsafe` block. The context for an iterator creates a nested `safe` block within the enclosing `unsafe` block.
- You can't use the `&` operator to take the address of a variable in an iterator method.

Before C# 13, iterators can't contain `unsafe` code (CS1629). Beginning with C# 13, this restriction is relaxed. All `yield return` statements must be in a safe context, but an iterator method can contain `unsafe` code.

ref safety in iterator methods

Iterator methods have special ref safety restrictions. These rules are relaxed in C# 13:

- **CS4013**: *Instance of type cannot be used inside a nested function, query expression, iterator block or async method*
- **CS8176**: *Iterators cannot have by-reference locals*

Before C# 13, iterators couldn't declare `ref` local variables. They could not declare any variables of a `ref struct` type.

Beginning with C# 13, `ref struct` types can be used in iterator methods, if they aren't accessed across `yield return` statement. Beginning with C# 13, iterator methods can declare `ref` local variables.

Compiler Error CS1638

Article • 04/12/2022

'identifier' is a reserved identifier and cannot be used when ISO language version mode is used

When the ISO language compatibility option is specified by the **/langversion** compiler switch, any identifier with double underscores anywhere in the identifier will produce this error. To avoid this error, eliminate any identifiers with double underscores, or do not use the ISO-1 language version option.

Example

The following sample generates CS1638:

C#

```
// CS1638.cs
// compile with: /langversion:ISO-1
class bad__identifier // CS1638 (double underscores are not ISO compliant)
{
}

// Try this instead:
//class GoodIdentifier
//{
//}

class CMain
{
    public static void Main() { }
}
```

See also

- **LangVersion** (C# Compiler Options)

Compiler Error CS1639

Article • 09/15/2021

The managed coclass wrapper class signature 'signature' for interface 'interface' is not a valid class name signature

This error occurs if the metadata for the interface is invalid. This should never occur if the interface was generated using C# or a supported .NET language. Check with the vendor of the interface or verify that the interface assembly was generated successfully.

Compiler Error CS1640

Article • 09/15/2021

foreach statement cannot operate on variables of type 'type' because it implements multiple instantiations of 'interface', try casting to a specific interface instantiation

The type inherits from two or more instances of IEnumerator<T>, which means there is not a unique enumeration of the type that `foreach` could use. Specify the type of IEnumerator<T> or use another looping construct.

Example

The following sample generates CS1640:

C#

```
// CS1640.cs

using System;
using System.Collections;
using System.Collections.Generic;

public class C : IEnumerable, IEnumerable<int>, IEnumerable<string>
{
    IEnumerator<int> IEnumerable<int>.GetEnumerator()
    {
        yield break;
    }

    IEnumerator<string> IEnumerable<string>.GetEnumerator()
    {
        yield break;
    }

    IEnumerator IEnumerable.GetEnumerator()
    {
        return (IEnumerator)((IEnumerable<string>)this).GetEnumerator();
    }
}

public class Test
{
    public static int Main()
    {
        foreach (int i in new C()){}    // CS1640

        // Try specifying the type of IEnumerable<T>
        // foreach (int i in (IEnumerable<int>)new C()){}
```

```
            return 1;
        }
    }
```

Compiler Error CS1643

Article • 09/15/2021

Not all code paths return a value in method of type 'type!'

This error occurs if a delegate body does not have a return statement, or has a return statement that the compiler is unable to verify will be reached. In the example below, the compiler does not attempt to predict the result of the branching condition in order to verify that the anonymous method block always returns a value.

Example

The following sample generates CS1643:

C#

```
// CS1643.cs
delegate int MyDelegate();

class C
{
    static void Main()
    {
        MyDelegate d = delegate
        {                   // CS1643
            int i = 0;
            if (i == 0)
                return 1;
        };
    }
}
```

Compiler Error CS1644

Article • 09/15/2021

Feature 'feature' is not part of the standardized ISO C# language specification, and may not be accepted by other compilers

This error occurs if you specified the **LangVersion** option ISO-1 and the code you are compiling uses features that are not part of the ISO 1.0 standard. To resolve this error, do not use any of the C# 2.0 compiler features such as generics or anonymous methods with the ISO-1 compatibility option.

The following sample generates CS1644:

C#

```
// CS1644.cs
// compile with: /langversion:ISO-1 /target:library
class C<T> {}   // CS1644
```

Compiler Error CS1646

Article • 09/15/2021

Keyword, identifier, or string expected after verbatim specifier: @

See string literals for the usage of the verbatim specifier '@'. The verbatim specifier is only allowed before a string, keyword or identifier. To resolve this error, remove the @ symbol from any inappropriate place or add the intended string, keyword or identifier.

The following sample generates CS1646:

C#

```
// CS1646
class C
{
   int i = @5;  // CS1646
   // Try this line instead:
   // int i = 5;
}
```

Compiler Error CS1647

Article • 09/15/2021

An expression is too long or complex to compile near 'code'

There was a stack overflow in the compiler processing your code. To resolve this error, simplify your code. If your code is valid, get help at the Developer Community site or StackOverflow.

Compiler Error CS1648

Article • 09/15/2021

Members of readonly field 'identifier' cannot be modified (except in a constructor or a variable initializer)

This error occurs when you attempt to modify a member of a field which is readonly where it is not allowed to be modified. To resolve this error, limit assignments to readonly fields to the constructor or variable initializer, or remove the readonly keyword from the declaration of the field.

Example

The following sample generates CS1648:

C#

```
// CS1648.cs
public struct Inner
{
    public int i;
}

class Outer
{
    public readonly Inner inner = new Inner();
}

class D
{
    static void Main()
    {
        var outer = new Outer();
        outer.inner.i = 1;  // CS1648
    }
}
```

Compiler Error CS1650

Article • 09/15/2021

Fields of static readonly field 'identifier' cannot be assigned to (except in a static constructor or a variable initializer)

This error occurs when you attempt to modify a member of a field which is readonly and static where it is not allowed to be modified. To resolve this error, limit assignments to readonly fields to the constructor or variable initializer, or remove the `readonly` keyword from the declaration of the field.

C#

```
// CS1650.cs
public struct Inner
{
    public int i;
}

class Outer
{
    public static readonly Inner inner = new Inner();
}

class D
{
    static void Main()
    {
        Outer.inner.i = 1;  // CS1650
    }
}
```

Compiler Error CS1654

Article • 09/15/2021

Cannot modify members of 'variable' because it is a 'read-only variable type'

This error occurs when you try to modify members of a variable which is read-only because it is in a special construct.

One common area that this occurs is within foreach loops. It is a compile-time error to modify the value of the collection elements. Therefore, you cannot make any modifications to elements that are value types, including structs. In a collection whose elements are reference types, you can modify accessible members of each element, but any try to add or remove or replace complete elements will generate Compiler Error CS1656.

Example

The following example generates error CS1654 because `Book` is a `struct`. To fix the error, change the `struct` to a class.

C#

```
using System.Collections.Generic;
using System.Text;

namespace CS1654
{

    struct Book
    {
        public string Title;
        public string Author;
        public double Price;
        public Book(string t, string a, double p)
        {
            Title=t;
            Author=a;
            Price=p;

        }
    }

    class Program
    {
        List<Book> list;
        static void Main(string[] args)
        {
```

```
            //Use a collection initializer to initialize the list
            Program prog = new Program();
            prog.list = new List<Book>();
            prog.list.Add(new Book ("The C# Programming Language",
                                    "Hejlsberg, Wiltamuth, Golde",
                                     29.95));
            prog.list.Add(new Book ("The C++ Programming Language",
                                    "Stroustrup",
                                     29.95));
            prog.list.Add(new Book ("The C Programming Language",
                                    "Kernighan, Ritchie",
                                    29.95));
            foreach(Book b in prog.list)
            {
                //Compile error if Book is a struct
                //Make Book a class to modify its members
                b.Price +=9.95; // CS1654
            }

        }
    }
}
```

Compiler Error CS1660

Article • 09/15/2021

Cannot convert anonymous method block to type 'type' because it is not a delegate type

This error occurs if you try to assign or otherwise convert an anonymous method block to a type which is not a delegate type.

The following sample generates CS1660:

C#

```
// CS1660.cs
delegate int MyDelegate();
class C {
   static void Main()
   {
     int i = delegate { return 1; };  // CS1660
     // Try this instead:
     // MyDelegate myDelegate = delegate { return 1; };
     // int i = myDelegate();
   }
}
```

Compiler Error CS1661

Article • 09/15/2021

Cannot convert anonymous method block to delegate type 'delegate type' because the specified block's parameter types do not match the delegate parameter types

This error occurs if, in an anonymous method definition, the parameter types of the anonymous method do not match the delegate parameter types. Check the number of parameters, the parameter types, and any ref or out parameters and verify an exact match.

The following sample generates CS1661:

C#

```
// CS1661.cs

delegate void MyDelegate(int i);

class C
{
    public static void Main()
    {
        MyDelegate d = delegate(string s) { };  // CS1661
    }
}
```

Compiler Error CS1662

Article • 09/15/2021

Cannot convert anonymous method block to delegate type 'delegate type' because some of the return types in the block are not implicitly convertible to the delegate return type

This error occurs if the anonymous method block's return statement had a type that was not implicitly convertible to the return type of the delegate.

The following sample generates CS1662:

C#

```
// CS1662.cs

delegate int MyDelegate(int i);

class C
{

  public static void Main()
  {
     MyDelegate d = delegate(int i) { return 1.0; };  // CS1662
     // Try this instead:
     // MyDelegate d = delegate(int i) { return (int)1.0; };
  }
}
```

Compiler Error CS1664

Article • 03/19/2024

Fixed size buffer of length 'length' and type 'type' is too big

The maximum size of a fixed-size buffer (as determined by the length multiplied by the element size) is 2^31 = 2147483648.

Compiler Error CS1667

Article • 09/15/2021

Attribute 'attribute' is not valid on property or event accessors. It is valid on 'declaration type' declarations only.

This error occurs if you use an attribute on a property or event accessor, when it should be on the property or event itself. This error could occur with the attributes CLSCompliantAttribute, ConditionalAttribute, and ObsoleteAttribute.

Example

The following sample generates CS1670:

C#

```
// CS1667.cs
using System;

public class C
{
    private int i;

    //Try this instead:
    //[Obsolete]
    public int ObsoleteProperty
    {
        [Obsolete]  // CS1667
        get { return i; }
        set { i = value; }
    }

    public static void Main()
    {
    }
}
```

Compiler Error CS1671

Article • 09/15/2021

A namespace declaration cannot have modifiers or attributes

Modifiers are not meaningful when applied to a namespace, so they are not allowed.

The following sample generates CS1671:

C#

```
// CS1671.cs
public namespace NS // CS1671
{

}
```

Compiler Error CS1672

Article • 09/15/2021

Invalid option 'option' for /platform; must be anycpu, x86, Itanium or x64

The options specify the processor type; use one of the forms listed.

Compiler Error CS1675

Article • 09/15/2021

Enums cannot have type parameters

To resolve this error, remove the type parameter from the `enum` declaration.

Example

The following sample generates CS1675:

C#

```
// CS1675.cs
enum E<T>  // CS1675
{
}

class CMain
{
    public static void Main()
    {
    }
}
```

Compiler Error CS1676

Article • 09/15/2021

Parameter 'number' must be declared with the 'keyword' keyword

This error occurs when the parameter type modifier in an anonymous method is different from that used in the declaration of the delegate you are casting the method to.

The following sample generates CS1676:

C#

```
// CS1676.cs
delegate void E(ref int i);
class Errors
{
   static void Main()
   {
      E e = delegate(out int i) { };   // CS1676
      // To resolve, use the following line instead:
      // E e = delegate(ref int i) { };
   }
}
```

Compiler Error CS1677

Article • 09/15/2021

Parameter 'number' should not be declared with the 'keyword' keyword

This error occurs when the parameter type modifier in an anonymous method does not match that used in the declaration of the delegate, to which you are casting the method.

Example

The following sample generates CS1677:

C#

```
// CS1677.cs
delegate void D(int i);
class Errors
{
    static void Main()
    {
        D d = delegate(out int i) { };   // CS1677
        // To resolve, use the following line instead:
        // D d = delegate(int i) { };

        D d = delegate(ref int j){}; // CS1677
        // To resolve, use the following line instead:
        // D d = delegate(int j){};
    }
}
```

Compiler Error CS1678

Article • 09/15/2021

Parameter 'number' is declared as type 'type1' but should be 'type2'

This error occurs when the parameter type in an anonymous method is different from the declaration of the delegate you are casting the method to.

The following sample generates CS1678:

C#

```
// CS1678
delegate void D(int i);
class Errors
{
   static void Main()
   {
      D d = delegate(string s) { };   // CS1678
      // To resolve, use the following line instead:
      // D d = delegate(int s) { };
   }
}
```

Compiler Error CS1679

Article • 09/15/2021

Invalid extern alias for '/reference'; 'identifier' is not a valid identifier

When using the external assembly alias feature of the **/reference** option, the text that follows **/reference:** and that precedes the '=' must be a valid C# identifier or keyword according to the C# Language Specification.

To correct this error, change text before the "=" to a valid C# identifier or keyword.

Example

The following example generates CS1679.

C#

```
// CS1679.cs
// compile with: /reference:123$BadIdentifier%=System.dll
class TestClass {
    static void Main()
    {
    }
}
```

Compiler Error CS1680

Article • 09/15/2021

Invalid reference alias option: 'alias=' -- missing filename.

This error occurs when you use the `alias` feature with the **/reference** compiler option without specifying a valid file name.

The following sample generates CS1680.

C#

```
// CS1680.cs
// compile with: /reference:alias=
// CS1680 expected
// To resolve, specify the name of a file with an assembly manifest
class MyClass {}
```

Compiler Error CS1681

Article • 09/15/2021

You cannot redefine the global extern alias

The global alias is already defined to include all unaliased references and therefore cannot be redefined.

Example

The following sample generates CS1681.

C#

```
// CS1681.cs
// compile with: /reference:global=System.dll
// CS1681 expected

// try this instead: /reference:System.dll
class A
{
   static void Main() {}
}
```

Compiler Error CS1688

Article • 09/15/2021

Cannot convert anonymous method block without a parameter list to delegate type 'delegate' because it has one or more out parameters

The compiler allows parameters to be omitted from an anonymous method block in most cases. This error arises when the anonymous method block does not have a parameter list, but the delegate has an `out` parameter. The compiler does not allow this situation because it would need to ignore the presence of the `out` parameter, which is unlikely to be the correct behavior.

Example

The following code generates error CS1688.

C#

```
// CS1688.cs
using System;
delegate void OutParam(out int i);
class ErrorCS1676
{
    static void Main()
    {
        OutParam o;
        o = delegate  // CS1688
        // Try this instead:
        // o = delegate(out int i)
        {
            Console.WriteLine("");
        };
    }
}
```

Compiler Error CS1689

Article • 09/15/2021

Attribute 'Attribute Name' is only valid on methods or attribute classes

This error only occurs with the **ConditionalAttribute** attribute. As the message states, this attribute can only be used on methods or attribute classes. For example, trying to apply this attribute to a class will generate this error.

Example

The following sample generates CS1689.

C#

```
// CS1689.cs
// compile with: /target:library
[System.Diagnostics.Conditional("A")]   // CS1689
class MyClass {}
```

Compiler Error CS1703

Article • 09/15/2021

An assembly with the same simple name 'name' has already been imported. Try removing one of the references or sign them to enable side-by-side.

The compiler removes references that have the same path and file name, but it is possible that the same file exists in two places, or that you forgot to change the version number. This error points out that two references have the same assembly identity and thus the compiler has no way of distinguishing between them in metadata. Either remove one of the redundant references, or make the references unique somehow, such as by incrementing the assembly version number.

The following code generates error CS1703.

Example 1

This code creates assembly A in the .\bin1 directory.

Save this example in a file named CS1703a1.cs, and compile it with the following flags: `/t:library /out:.\bin1\cs1703.dll /keyfile:key.snk`

```
C#
```

```
using System;
public class A { }
```

Example 2

This code creates a copy of assembly A in the .\bin2 directory.

Save this example in a file named CS1703a2.cs, and compile it with the following flags: `/t:library /out:.\bin2\cs1703.dll /keyfile:key.snk`

```
C#
```

```
using System;
public class A { }
```

Example 3

This code references the assembly A in the two prior modules.

Save this example in a file named CS1703ref.cs, and compile it with the following flags: `/t:library /r:A2=.\bin2\cs1703.dll /r:A1=.\bin1\cs1703.dll`

C#

```
extern alias A1;
extern alias A2;
```

Compiler Error CS1705

Article • 09/15/2021

Assembly 'AssemblyName1' uses 'TypeName' which has a higher version than referenced assembly 'AssemblyName2'

You are accessing a type that has a higher version number than the version number in a referenced assembly. Typically, this error is caused by the accidental use of two versions of the same assembly.

For example, suppose that you have two assemblies, Asmb1 and Asmb2. Assembly Asmb1 references version 1.0 of assembly Asmb2. Assembly Asmb1 also uses a class `MyClass` that was added to assembly Asmb2 in version 2.0. The compiler has unification rules for binding references, and a reference to `MyClass` in version 2.0 cannot be satisfied by version 1.0.

Examples

The following more detailed example consists of four code modules:

- Two DLLs that are identical except for a version attribute.
- A third DLL that references the first two.
- A client that references only version 1.0 of the identical DLLs, but accesses a class from version 2.0.

The following code creates the first of the identical DLLs. For information about how to generate a key file, see **KeyFile** (C# Compiler Options).

C#

```
// CS1705a.cs

// Compile by using the following command:
//      csc /target:library /out:C:\\CS1705.dll /keyfile:mykey.snk
CS1705a.cs

// The DLL is created in the C:\ directory.

// The AssemblyVersion attribute specifies version 1.0 for this DLL.

[assembly:System.Reflection.AssemblyVersion("1.0")]
public class Class1
{
```

```
    public void Method1() {}
}
```

The following code defines version 2.0 of the assembly, as specified by the AssemblyVersionAttribute attribute.

C#

```
// CS1705b.cs

// Compile by using the following command:
//     csc /target:library /out:CS1705.dll /keyfile:mykey.snk CS1705b.cs

// The DLL is created in the current directory.

// The AssemblyVersion attribute specifies version 2.0 for this DLL.

[assembly:System.Reflection.AssemblyVersion("2.0")]
public class Class1
{
    public void Method1() { }
}
```

The following code references the two DLL versions that are defined in the preceding code. `AssemblyA` refers to the DLL created by CS1705a.cs (version 1.0). `AssemblyB` refers to the DLL created by CS1705b.cs (version 2.0). In `ClassC`, two methods are defined. The first, `Return1A`, returns an object of type `Class1A`, which is an alias for `Class1` from version 1.0 of the DLL. The second, `Return1B`, returns an object of type `Class1B`, which is an alias for `Class1` from version 2.0 of the DLL. The definition of `Return1A` creates a dependency on version 1.0; the definition of `Return1B` creates a dependency on version 2.0.

C#

```
// CS1705c.cs

// Compile by using the following command. AssemblyA refers to the DLL
created by
// CS1705a.cs (version 1.0). AssemblyB refers to the DLL created by
CS1705b.cs
// (version 2.0).
//    csc /target:library /r:AssemblyA=C:\\CS1705.dll
/r:AssemblyB=CS1705.dll CS1705c.cs

extern alias AssemblyA;
extern alias AssemblyB;

// Class1A is an alias for type Class1 from VS1705a.cs, which is in version
```

```
1.0
// of the assembly. Class1B is an alias for type Class1 from CS1705b.cs,
which
// is in version 2.0 of the assembly.

using Class1A = AssemblyA::Class1;
using Class1B = AssemblyB::Class1;

// Method Return1A in ClassC returns an object of type Class1A, which is
// Class1 from version 1.0 of the DLL. Method Return1B returns an object
// of type Class1B, which is Class1 from version 2.0 of the DLL.

public class ClassC
{
    // The following line creates a dependency on version 1.0 of CS1705.dll.
    // This is not the source of the problem when ClassC is accessed from
    // CS1705d.cs because CS1705d.cs references version 1.0 of the DLL.
    // Therefore, type Class1A and the assembly have the same version.
    public static Class1A Return1A() { return new Class1A(); }

    // The following line creates a dependency on version 2.0 of CS1705.dll.
    // This causes compiler error CS1705 when ClassC is accessed from
    // CS1705d.cs, because CS1705d.cs does not reference version 2.0 of the
    // DLL. Class1B is the alias for Class1 in version 2.0, and CS1705d.cs
    // references version 1.0.
    public static Class1B Return1B() { return new Class1B(); }
}
```

The following code generates compiler error CS1705. It references the DLL created by CS1705a.cs (version 1.0). However, in the `Main` method, the code accesses `ClassC` from CS1705c.cs. `ClassC` uses a type that is defined in CS1705b.cs (version 2.0). This causes compiler error CS1705 because the type has a version number for CS1705.dll that is higher than the referenced version of CS1705.dll.

C#

```
// CS1705d.cs

// Compile by using the following command:
//     csc /reference:C:\\CS1705.dll /reference:CS1705c.dll CS1705d.cs

// C:\\CS1705.dll is version 1.0 of the assembly.

class Tester
{
    static void Main()
    {
        // Return1A returns a type defined in version 1.0.
        ClassC.Return1A().Method1();
        // Return1B returns a type defined in version 2.0.
        ClassC.Return1B().Method1();
```

```
    }
}
```

You can resolve the error in one of the following ways:

- Update the code so that all files use the same version of the DLL.
- Add a reference to version 2.0 of the DLL to CS1705d.cs by using the following command to compile:

  `csc /reference:C:\\CS1705.dll /reference:CS1705.dll /reference:CS1705c.dll CS1705d.cs`

  Although the program compiles when you use this command, it still does not run. To enable the program to run, you can provide an application configuration file that includes a <dependentAssembly> element that uses <assemblyIdentity> and <codeBase> child elements to specify the location of version 1.0 of the DLL. For more information about configuration files, see Configuring Apps.

See also

- extern alias
- :: Operator

Compiler Error CS1713

Article • 09/15/2021

Unexpected error building metadata name for type Typename1—'Reason'

This error is often caused by an internal compiler error. Making some minor changes to your code, such as shortening the length of the name, and then recompiling, may resolve this error.

Compiler Error CS1715

Article • 09/15/2021

'Type1': type must be 'Type2' to match overridden member 'MemberName'

This error is the same as Compiler Error CS0508, except that CS0508 now only applies to methods that have return types, while CS1715 applies to properties and indexers that only have 'types' instead of 'return types'.

Example

The following code generates CS1715.

C#

```
// CS1715.cs
abstract public class Base
{
    abstract public int myProperty
    {
        get;
        set;
    }
}

public class Derived : Base
{
    int myField;
    public override double myProperty  // CS1715
    // try the following line instead
    // public override int myProperty
    {
        get { return myField; }
        set { myField;= value; }
    }

    public static void Main()
    {
        Derived d = new Derived();
        d.myProperty = 5;
    }
}
```

Compiler Error CS1719

Article • 09/15/2021

Error reading Win32 resource file 'File Name' -- 'reason'

An attempt to read the Win32 resource file failed for the reason given in the error, typically something like "file not found" or "access denied." This error is resolved by correcting the problem described by the reason.

Compiler Error CS1721

Article • 09/15/2021

Class 'class' cannot have multiple base classes: 'class_1' and 'class_2'

The most common cause of this error message is attempting to use multiple inheritance. A class in C# may only inherit directly from one class. However, a class can implement any number of interfaces.

Example

The following example shows one way in which CS1721 is generated:

C#

```
// CS1721.cs
public class A {}
public class B {}
public class MyClass : A, B {}   // CS1721
```

To correct this error

The following are different ways to correct this error:

- Make class `B` inherit from `A`, and `MyClass` inherit from `B`:

  C#

  ```
  public class A {}
  public class B : A {}
  public class MyClass : B {}
  ```

- Declare `B` as an interface. Make `MyClass` inherit from the interface `B`, and the class `A`:

  C#

  ```
  public class A {}
  public interface B {}
  public class MyClass : A, B {}
  ```

See also

- Polymorphism
- Interfaces

Compiler Error CS1722

Article • 09/15/2021

Base class 'class' must come before any interfaces

When specifying a class to inherit from and interfaces to implement, the class name must be specified first.

Example

The following sample generates CS1722.

C#

```
// CS1722.cs
// compile with: /target:library
public class A {}
interface I {}

public class MyClass : I, A {}   // CS1722
public class MyClass2 : A, I {}   // OK
```

Compiler Error CS1724

Article • 09/15/2021

Value specified for the argument to 'System.Runtime.InteropServices.DefaultCharSetAttribute' is not valid

This error is generated by an invalid argument to the DefaultCharSetAttribute class.

Example

The following example generates CS1724.

C#

```
// CS1724.cs
using System.Runtime.InteropServices;
// To resolve, replace 42 with a valid CharSet value.
 [module:DefaultCharSetAttribute((CharSet)42)]   // CS1724
class C {

[DllImport("F.Dll")]
extern static void FW1Named();

static void Main() {}
}
```

Compiler Error CS1725

Article • 09/15/2021

Friend assembly reference 'reference' is invalid. InternalsVisibleTo declarations cannot have a version, culture, public key token, or processor architecture specified.

You cannot add a version culture in a friend assembly reference. Partial classes should be visible to friend assemblies.

Example

The following sample generates CS1725.

C#

```
// CS1725.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("partial01,version=1.1.0.0")]   // CS1725
// try the following line instead
// [assembly:InternalsVisibleTo("partial01")]

partial class TestClass
{
   public static string strBar = "my string";
}
```

See also

- How to: Create Signed Friend Assemblies

Compiler Error CS1726

Article • 09/15/2021

Friend assembly reference 'reference' is invalid. Strong-name signed assemblies must specify a public key in their InternalsVisibleTo declarations.

A strong name signed assembly can only grant friend assembly access, made with the InternalsVisibleToAttribute, to other strongly signed assemblies.

To resolve CS1726, either sign (give a strong name to) the assembly to which you want to grant friend access, or don't grant friend access.

For more information, see Friend Assemblies.

Example

The following sample generates CS1726.

C#

```
// Save this code as CS1726.cs

// Run the following command to create CS1726.key:
//      sn -k CS1726.key

// Then compile by using the following command:
//      csc /keyfile:CS1726.key /target:library CS1726.cs

using System.Runtime.CompilerServices;

// The following line causes compiler error CS1726.
[assembly: InternalsVisibleTo("UnsignedAssembly")]

// To get rid of the error, try the following line instead.
//[assembly: InternalsVisibleTo("SignedAssembly,
PublicKey=0024000004800000940000000602000000240000525341310004000001000100031d7b6f3abc16c7de526fd67ec2926fe68ed2f9901afbc5f1b6b428bf6cd9086021a0b38b76bc340dc6ab27b65e4a593fa0e60689ac98dd71a12248ca025751d135df7b98c5f9d09172f7b62dabdd302b2a1ae688731ff3fc7a6ab9e8cf39fb73c60667e1b071ef7da5838dc009ae0119a9cbff2c581fc0f2d966b77114b2c4")]

class A { }
```

See also

- How to: Create Signed Friend Assemblies

Compiler Error CS1727

Article • 09/15/2021

Cannot send error report automatically without authorization. Please visit '' to authorize sending error report.

The Web site listed in the error text explains how to enable automatic error reporting for Visual Studio 2005 command line tools.

Example

The following sample generates CS1727.

C#

```
// CS1727.cs
// compile with: /errorreport:send
// CS1727 expected
class Test
{
    static void Main(){}
}
```

Compiler Error CS1728

Article • 09/15/2021

Cannot bind delegate to 'member' because it is a member of 'type'

You cannot bind delegates to members of `Nullable` value types.

Example

The following sample generates CS1728:

C#

```
// CS1728.cs
class Test
{
   delegate T GetT<T>();
   delegate T GetT1<T>(T t);

   delegate bool E(object o);
   delegate int I();
   delegate string S();

   static void Main()
   {
      int? x = null;
      int? y = 5;

      GetT<int> d1 = x.GetValueOrDefault;   // CS1728
      GetT<int> d2 = y.GetValueOrDefault;   // CS1728
      GetT1<int> d3 = x.GetValueOrDefault;   // CS1728
      GetT1<int> d4 = y.GetValueOrDefault;   // CS1728
   }
}
```

Compiler Error CS1729

Article • 09/15/2021

'type' does not contain a constructor that takes 'number' arguments.

This error occurs when you either directly or indirectly invoke the constructor of a class but the compiler cannot find any constructors with the same number of parameters. In the following example, the `test` class has no constructors that take any arguments. It therefore has only a parameterless constructor that takes zero arguments. Because in the second line in which the error is generated, the derived class declares no constructors of its own, the compiler provides a parameterless constructor. That constructor invokes a parameterless constructor in the base class. Because the base class has no such constructor, CS1729 is generated.

To correct this error

1. Adjust the number of parameters in the call to the constructor.
2. Modify the class to provide a constructor with the parameters you must call.
3. Provide a parameterless constructor in the base class.

Example

The following example generates CS1729:

C#

```
// cs1729.cs
class Test
{
    static int Main()
    {
        // Class Test has only a parameterless constructor, which takes no
arguments.
        Test test1 = new Test(2); // CS1729
        // The following line resolves the error.
        Test test2 = new Test();

        // Class Parent has only one constructor, which takes two int
parameters.
        Parent exampleParent1 = new Parent(10); // CS1729
        // The following line resolves the error.
        Parent exampleParent2 = new Parent(10, 1);
```

```
        return 1;
    }
}

public class Parent
{
    // The only constructor for this class has two parameters.
    public Parent(int i, int j) { }
}

// The following declaration causes a compiler error because class Parent
// does not have a constructor that takes no arguments. The declaration of
// class Child2 shows how to resolve this error.
public class Child : Parent { } // CS1729

public class Child2 : Parent
{
    // The constructor for Child2 has only one parameter. To access the
    // constructor in Parent, and prevent this compiler error, you must
provide
    // a value for the second parameter of Parent. The following example
provides 0.
    public Child2(int k)
        : base(k, 0)
    {
        // Add the body of the constructor here.
    }
}
```

See also

- Inheritance
- Constructors

Compiler Error CS1730

Article • 03/15/2023

Assembly and module attributes must precede all other elements defined in a file except using clauses and extern alias declarations.

An attribute applied at the assembly level cannot appear after any type definitions.

To correct this error

1. Move the attribute to the top of the file, but below the `using` directives and `extern` alias declarations.

Example

The following code generates CS1730:

C#

```
// cs1730.cs
class Test
{
}
[assembly: System.Attribute] // CS1730
```

See also

- Attributes

Compiler Error CS1731

Article • 09/15/2021

Cannot convert 'expression' to delegate because some of the return types in the block are not implicitly convertible to the delegate return type.

This error is generated when a lambda expression or anonymous method has a return type that is not compatible with the delegate's return type.

To correct this error

1. Change the return type of either the delegate or the expression.

Example

The following code generates CS1731:

C#

```
class CS1731
{
    delegate double D();
    D d = () => { return "Who knows the real sword of Gryffindor?"; };
}
```

Compiler Error CS1732

Article • 09/15/2021

Expected parameter.

This error is produced when a lambda expression contains a comma following an input parameter but does not specify the following parameter.

To correct this error

1. Either remove the comma, or add the input parameter that the compiler expects to find after the comma.

Example

The following example produces CS1732:

C#

```
// cs1732.cs
// compile with: /target:library
class Test
    {
        delegate void D(int x, int y);
        static void Main()
        {
        D d = (x,) => { }; // CS1732
      }
    }
```

Compiler Error CS1733

Article • 09/15/2021

Expected expression.

This error is produced whenever the compiler is expecting an expression on the line where the error occurred. In the following example, the trailing comma in the initializer indicates to the compiler that another expression will follow.

To correct this error

- Provide the missing expression.
- Remove the tokens that are causing the compiler to expect an expression.

Example

The following example produces CS1733 because of the trailing comma:

C#

```
// cs1733.cs
using System.Collections.Generic;
public class Test
{
    public static void Main()
    {
        List<int> list = new List<int>() {{ 1, 2, }};// CS1733
    }
}
```

Compiler Error CS1736

Article • 09/17/2022

Default parameter value for must be a compile-time constant

Example

The following sample generates CS1736:

```
C#
```

```
// CS1736.cs

public unsafe class C
{
    static void F(int i = G())
    {
        // ...
    }
    static int G() => 0;
```

A default parameter value is evaluated upon the invocation of the method. What a value may be when the method is eventually invoked cannot be pre-determined at declaration-time unless that value is constant at compile-time.

To correct this error

If a dynamically evaluated value is required, consider using a compile-time constant as a marker value which is then checked at run-time:

```
C#
```

```
    static void F(int i = -1)
    {
        if(i == -1) i = G();
        //...
    }
```

Compiler Error CS1737

Article • 10/07/2022

Optional parameters must appear after all required parameters

The compiler does not support optional parameters being declared before required parameters. All optional parameters must be after all required parameters.

Example

The following sample generates CS1737:

C#

```
// CS1737.cs (7,45)
class C
{
    static void F(object? x)
    {
        G(y: x);
    }
    static void G(object? x = null, object y)
    {
    }
}
```

To correct this error

The signature for this method may be changed without effecting existing code that calls the method because a value for the optional parameter has not been used. For example:

C#

```
// CS1737.cs (7,45)
class C
{
    static void F(object? x)
    {
        G(y: x);
    }
    static void G(object y, object? x = null)
    {
    }
}
```

Compiler Error CS1750

Article • 09/09/2023

A value of type cannot be used as a default parameter because there are no standard conversions to type

Example

The following sample generates CS1750:

```
C#
```

```
public struct S
{
    public override string ToString() { return "S::ToString"; }
}
public class A
{
    public static S Goo(S p = 42) { return p; }
}
```

There is no standard conversion between `int` and the newly declared struct `S`. Using an `int` compile-time constant to initialize an instance of struct `S` results in CS1750. Adding a user-defined conversion operator (for example, `public static implicit operator S(int n) => ...`) will not correct this error because that does not add a *standard conversion*.

Compiler Error CS1763

Article • 09/20/2022

A default parameter value of a reference type other than string can only be initialized with null

Example

The following sample generates CS1763:

```
C#
```

```
// CS1763.cs (0,0)
class Program
{
    public void Goo<T, U>(T t = default(U)) where U : T
    {
    }
    static void Main(string[] args)
    {

    }
}
```

This example generates CS1763 because the `Goo<T,U>` parameter is declared with a default value of `default(U)` when the type of the parameter is `T`, despite the constraint that `U` derive from base class `T`.

To correct this error

Changing `default(U)` to use the corresponding type argument corrects this error:

```
C#
```

```
    public void Goo<T, U>(T t = default(T)) where U : T
    {
    }
```

Compiler Error CS1900

Article • 11/14/2023

Warning level must be in the range 0-4

The **WarningLevel** compiler option can only take one of five possible values (0, 1, 2, 3, or 4). Any other value passed to **/warn** will result in CS1900.

The following sample generates CS1900:

C#

```
// CS1900.cs
// compile with: /W:5
// CS1900 expected
class x
{
   public static void Main()
   {
   }
}
```

> **Note**
>
> The compiler no longer generates this error. Values greater than 4 represent "warning waves" and are valid, even if no warnings are defined for that wave yet.

Compiler Error CS1902

Article • 09/15/2021

Invalid option 'option' for /debug; must be full or pdbonly

An invalid option was passed to the **DebugType** compiler option.

The following sample generates CS1902:

C#

```
// CS1902.cs
// compile with: /debug:x
// CS1902 expected
class x
{
   public static void Main()
   {
   }
}
```

Compiler Error CS1906

Article • 09/15/2021

Invalid option 'option'; Resource visibility must be either 'public' or 'private'

This error indicates an invalid **Resources** (Embed Resource File to Output) or **LinkResources** (Link to .NET Framework Resource) command line option. Check the syntax of the **/resource** or **/linkresource** command line option, and make sure that the accessibility modifier used is either **public** or `private`.

Compiler Error CS1908

Article • 10/03/2023

The type of the argument to the DefaultParameterValue attribute must match the parameter type

This error is generated when you pass a value of the wrong type to DefaultParameterValueAttribute. Ensure the type of the attribute argument matches that of the target parameter.

Example

The following sample generates CS1908:

C#

```
// CS1908.cs
// compile with: /target:library
using System.Runtime.InteropServices;

public interface ISomeInterface
{
    void Bad([DefaultParameterValue("true")] bool b);   // CS1908
    void Good([DefaultParameterValue(true)] bool b);   // OK
}
```

Compiler Error CS1909

Article • 10/03/2023

The DefaultParameterValue attribute is not applicable on parameters of type 'type'

CS1909 is generated when you use a DefaultParameterValueAttribute attribute that is not applicable on the parameter type. This error message is no longer emitted, see CS1910 for more information.

Compiler Error CS1910

Article • 10/03/2023

Argument of type 'type' is not applicable for the DefaultParameterValue attribute

For parameters whose type is object, the argument of the DefaultParameterValueAttribute must be `null`, an integral type, a floating point, `bool`, `string`, `enum`, or `char`. The argument can not be of type Type or any array type.

Example

The following sample generates CS1910:

C#

```
// CS1910.cs
// compile with: /target:library
using System.Runtime.InteropServices;

public interface ISomeInterface
{
    void Bad1([DefaultParameterValue(typeof(object))] object o);   // CS1910
    void Bad2([DefaultParameterValue(new int[] { 1, 2 })] int[] arr);   //
CS1910
}
```

Compiler Error CS1912

Article • 09/15/2021

Duplicate initialization of member 'name'.

An object initializer can initialize each member only one time.

To correct this error

1. Remove the second initialization of the member in the object initializer.

Example

The following code generates CS1912 because `memberA` is initialized two times:

C#

```
// cs1912.cs
using System.Linq;

public class TestClass
{
    public int memberA { get; set; }
    public int memberB { get; set; }
}

public class Test
{
    static void Main()
    {
        TestClass tc = new TestClass() { memberA = 5, memberA = 6, memberB =
2}; // CS1912
    }
}
```

See also

- Object and Collection Initializers

Compiler Error CS1913

Article • 10/27/2021

Member 'name' cannot be initialized. It is not a field or property.

Object initializers can only be used to initialize accessible fields or properties.

To correct this error

1. Initialize the class member in a regular constructor or other initialization method.

Example

The following example generates CS1913:

C#

```
// cs1912.cs
class A
{
    public delegate void D();
    public event D myEvent;
}

public class Test
{
    static void Main()
    {

        A a = new A() {myEvent = M}; // CS1913
    }

    public void M(){}
}
```

See also

- The C# type system

Compiler Error CS1914

07/09/2025

Static field 'name' cannot be assigned in an object initializer

Object initializers by definition initialize objects, or instances, of classes. They cannot be used to initialize a `static` field of a type. No matter how many instances of a class are created, there is only one copy of a `static` field. For generic types, each closed generic type has its own copy of static fields. Static fields marked with ThreadStaticAttribute have one copy per thread.

To correct this error

1. Either change the field to an instance field in the type, or remove the attempt to initialize it from the object initializer.

Example

The following code generates CS1914 because the initializer tries to initialize the `TestClass.Number` field, which is `static`:

C#

```
// cs1914.cs
using System.Linq;
public class TestClass
{
    public string Message { get; set; }
    public static int Number { get; set; }
}
class Test
{
    static void Main()
    {
        TestClass b = new TestClass() { Message = "Hello", Number = "555-1212" };
// CS1914

    }
}
```

Compiler Error CS1917

Article • 09/15/2021

Members of read-only field 'name' of type 'struct name' cannot be assigned with an object initializer because it is of a value type.

Read-only fields that are value types can only be assigned in a constructor.

To correct this error

- Change the struct to a class type.
- Initialize the struct with a constructor.

Example

The following code generates CS1917:

C#

```
// cs1917.cs
class CS1917
{
    public struct TestStruct
    {
        public int i;
    }
    public class C
    {
        public readonly TestStruct str = new TestStruct();
        public static int Main()
        {
            C c = new C { str = { i = 1 } }; // CS1917
            return 0;
        }
    }
}
```

Compiler Error CS1918

Article • 09/15/2021

Members of property 'name' of type 'type' cannot be assigned with an object initializer because it is of a value type.

This error occurs when you try to use an object initializer to initialize the properties of a struct type that is itself a property of the class that is being initialized.

To correct this error

1. If you must fully initialize the fields of the property in the object initializer, change the struct to a class type. Otherwise, initialize the struct members in a separate method call after you create the object by using the object initializer.

Example

The following example generates CS1918:

C#

```
// cs1918.cs
public struct MyStruct
{
    public int i;

}
public class Test
{
    private MyStruct str = new MyStruct();
    public MyStruct Str
    {
        get
        {
            return str;
        }
    }
    public static int Main()
    {
        Test t = new Test { Str = { i = 1 } }; // CS1918
        return 0;
    }
}
```

See also

- Object and Collection Initializers

Compiler Error CS1922

Article • 09/15/2021

Collection initializer requires its type 'type' to implement System.Collections.IEnumerable.

In order to use a collection initializer with a type, the type must implement `IEnumerable`. This error can occur if you accidentally use collection initializer syntax when you meant to use an object initializer.

To correct this error

- If the type does not represent a collection, use object initializer syntax instead of collection initializer syntax.
- If the type does represent a collection, modify it to implement `IEnumerable` before you can use collection initializers to initialize objects of that type.
- If the type represents a collection and you do not have access to the source code, just initialize its elements by using their class constructors or other initialization methods.

Example

The following code produces CS1922:

C#

```
// cs1922.cs
public class Test
{
    public static void Main()
    {
        // Collection initializer.
        var tc = new TestClass  {1,"hello"} ; // CS1922

        // Object initializer.
        var tc2 = new TestClass { memberA = 1, memberB = "hello" }; // OK
    }
}

public class TestClass
{
    public int memberA { get; set; }
```

```
    public string memberB { get; set; }
}
```

See also

- Object and Collection Initializers

Compiler Error CS1926

Article • 09/15/2021

Error reading Win32 manifest file 'filename' -- 'error'.

This error is generated when the following conditions are true:

1. The **/win32manifest** option is specified either on the command line or by right-clicking the **Project** icon in **Solution Explorer**, pointing to **Add**, clicking **New Item**, and then clicking **Application Manifest File**.

2. The file is either corrupted or missing.

To correct this error

1. Remove the option.

2. Replace, repair, or regenerate the file.

Example

The following example generates CS1926 when it is compiled with a corrupted for missing win32 manifest file:

C#

```
// cs1926.cs
// Compile with: /win32manifest: ../../app.manifest
// CS1926
class Test
{
    public static int Main()
    {
        return 1;
    }
}
```

See also

- **Win32Manifest** (C# Compiler Options)

Compiler Error CS1930

Article • 09/15/2021

The range variable 'name' has already been declared

The range variable in a query expression is in scope until the query expression terminates. It must therefore have a unique identifier.

To correct this error

1. Give a unique name to each range variable that is introduced in a query expression.

Example

The following example generates CS1930 because the identifier `num` is used for the range variable in the `from` clause and for the range variable introduced by the `let` clause.

C#

```
// cs1930.cs
using System.Linq;
class Program
{
    static void Main()
    {
        int[] nums = { 0, 1, 2, 3, 4, 5 };
        var query = from num in nums
                    let num = 3 // CS1930
                    select num;
    }
}
```

See also

- LINQ Query Expressions

Compiler Error CS1931

Article • 09/15/2021

The range variable 'variable' conflicts with a previous declaration of 'variable'.

The declaration of a range variable, just like every other declaration, must have an identifier which is unique within the variable's declaration space.

To correct this error

1. Give the range variable a unique name.

Example

The following code generates CS1931 because the identifier `x` is used both as a local variable in `Main` and as the range variable in the query expression:

C#

```
// cs1931.cs
class Test
    {
        static void Main()
        {
            int x = 1;
            var y = from x in Enumerable.Range(1, 100) // CS1931
                    select x;
        }
    }
```

See also

- LINQ Query Expressions

Compiler Error CS1932

Article • 09/15/2021

Cannot assign 'expression' to a range variable.

The compiler must be able to infer the type of a range variable, whether it is introduced in a `from` clause or a `let` clause. It cannot be null because null is not a type, and it cannot be assigned with an expression of an unsafe type.

To correct this error

- Remove the assignment that is not valid.
- Explicitly cast the expression to an allowed type

Example

The following code generates CS1932 because the type of the range variable cannot be inferred. Cast the value to the intended type to fix the error, as shown in the following example.

C#

```
// CS1932.cs
using System.Linq;
class Test
{
    static void Main()
    {

        var x = from i in Enumerable.Range(1, 100)
                let k = null // CS1932
                // Try the following line instead.
                let k = (string) null
                select i;
    }
}
```

See also

- LINQ Query Expressions

Compiler Error CS1933

Article • 09/15/2021

Expression cannot contain query expressions

Some variables cannot be initialized with a query expression. Constants cannot be initialized with query expressions because constants may only be initialized with some combination of literals, named constants, and mathematical operators.

To correct this error

1. Remove the modifier from the query variable.

Example

The following example generates CS1933:

C#

```
// cs1933.cs
using System.Linq;
using System.Collections;

class Program
{
    const IEnumerable e = from x in new[] { 1, 2, 3 } select x; // CS1933
    static int Main()
    {
        return 1;
    }
}
```

> **Note**
>
> This compiler error is no longer used in Roslyn. The previous example generates CS0133 when compiled with Roslyn.

See also

- LINQ in C#

Compiler Error CS1934

Article • 09/15/2021

Could not find an implementation of the query pattern for source type 'type'. 'method' not found. Consider explicitly specifying the type of the range variable 'name'.

This error is produced if a query expression specifies a data source for which no standard query operators are implemented. One way to produce this error is to specify an `ArrayList` without giving an explicit type for the range variable.

To correct this error

1. In the following example, the solution is to just specify the type of the range variable:

   ```
   C#

   var q = from int x in list
   ```

Example

The following example shows one way to produce CS1934:

```
C#

// cs1934.cs
using System.Linq;
using System.Collections;
static class Test
{
    public static void Main()
    {
        var list = new ArrayList { 0, 1, 2, 3, 4, 5 };
        var q = from x in list // CS1934
                select x + 1;
    }
}
```

See also

- How to query an ArrayList with LINQ

Compiler Error CS1935

Article • 09/15/2021

Could not find an implementation of the query pattern for source type 'type'. 'method' not found. Are you missing a using directive for 'System.Linq'?

The source type in a query must be `IEnumerable`, `IEnumerable<T>`, or a derived type, or a type for which you or someone else has implemented the standard query operators. If the source type is an `IEnumerable` or `IEnumerable<T>`, you must add a `using` directive for the `System.Linq` namespace in order to bring the standard query operator extension methods into scope. Custom implementations of the standard query operators must be brought into scope in the same way, with a `using` directive and, if necessary, a reference to the assembly.

To correct this error

Add the required `using` directives and references to the project.

Example

The following code generates CS1935 because the `using` directive for System.Linq is commented out:

C#

```
// cs1935.cs
// CS1935
using System;
using System.Collections.Generic;
// using System.Linq;

class Test
{
    static int Main()
    {
        int[] nums = { 0,1,2,3,4,5 };
        IEnumerable<int> e = from n in nums
                             where n > 3
                             select n;
        return 0;
    }
}
```

See also

- Standard Query Operators Overview

Compiler Error CS1936

Article • 09/15/2021

Could not find an implementation of the query pattern for source type 'type'. 'method' not found.

In order to query a source type, that type must implement the standard query operator methods that you are invoking in the query. The implementation can be either in the form of class members or extension methods that are brought into scope with the appropriate `using` directive.

To correct this error

- Make sure that you are querying a collection of objects, not an individual object.
- Make sure that you have specified the necessary `using` directives.

Example

The following example produces CS1936:

C#

```
// cs1936.cs
using System.Collections;
using System.Linq;
class Test
{
    static int Main()
    {
        object obj;
        IEnumerable e = from x in obj // CS1936
                        select x;
        return 0;
    }
}
```

This error typically occurs when you accidentally try to query an object of some type instead of a collection of those objects.

See also

- Standard Query Operators Overview

Compiler Error CS1937

Article • 09/15/2021

The name 'name' is not in scope on the left side of 'equals'. Consider swapping the expressions on either side of 'equals'.

The `equals` keyword is a special operator that is used in a `join` clause to determine equality between two expressions. The range variable for the left side source sequence is in scope on the left side of equals, and the range variable for the right side source is only in scope on the left side of equals. You can verify this by experimenting with IntelliSense in the following code example.

To correct this error

1. Swap the position of the two range variables as shown in the commented line in the following example:

Example

The following example generates CS1937.

C#

```
// cs1937.cs
using System.Linq;
class Test
{
    static void Main()
    {
        int[] sourceA = { 1, 2, 3, 4, 5 };
        int[] sourceB = { 3, 4, 5, 6, 7 };

        var query = from a in sourceA
                    join b in sourceB on b equals a // CS1937
                    // Try the following line instead.
                    //join b in sourceB on a equals b
                    select new { a, b };
    }
}
```

The left side is generally called the "outer" side and the right is generally called the "inner" side.

See also

- join clause

Compiler Error CS1938

Article • 09/15/2021

The name 'name' is not in scope on the right side of 'equals'. Consider swapping the expressions on either side of 'equals'.

The `equals` keyword is a special operator that is used in a `join` clause to determine equality between two expressions. The range variable for the left side source sequence is in scope on the left side of equals, and the range variable for the right side source is only in scope on the left side of equals. You can verify this by experimenting with IntelliSense in the following code example.

To correct this error

1. Swap the position of the two range variables as shown in the commented line in the following example:

Example

The following code generates CS1938:

C#

```
// cs1938.cs
using System.Linq;
class Test
{
    static void Main()
    {
        int[] sourceA = { 1, 2, 3, 4, 5 };
        int[] sourceB = { 3, 4, 5, 6, 7 };

        var query = from a in sourceA
                    join b in sourceB on b equals a // CS1938
                    // Try the following line instead.
                    // join b in sourceB on a equals b
                    select new { a, b };
    }
}
```

See also

- join clause

Compiler Error CS1940

Article • 09/15/2021

Multiple implementations of the query pattern were found for source type 'type'. Ambiguous call to 'method'.

This error is generated when multiple implementations of a query method are defined and the compiler cannot disambiguate which one is best to use for the query. In the following example, both versions of `Select` have the same signature, because they both accept one `int` as an input parameter and have `int` as a return value.

To correct this error

1. Provide only one implementation for each method.

Example

The following code generates CS1940:

C#

```
// cs1940.cs
using System; //must include explicitly for types defined in 3.5
class Test
{
    public delegate int Dele(int x);
    int num = 0;
    public int Select(Func<int, int> d)
    {
        return d(this.num);
    }
    public int Select(Dele d) // CS1940
    {
        return d(this.num) + 1;
    }
    public static void Main()
    {
        var q = from x in new Test()
        select x;
    }
}
```

See also

- Standard Query Operators Overview

Compiler Error CS1941

Article • 09/15/2021

The type of one of the expressions in the 'clause' clause is incorrect. Type inference failed in the call to 'method'.

Type inference in query expressions flows from the type of the elements in the data source(s).

To correct this error

1. If it is not immediately obvious why the error is occurring, examine the query carefully and trace the type of the result of each clause from the data source to the point where the error is occurring.

Example

The following code generates CS1941 because the `equals` operator is being asked to compare an `int` to a `string`.

C#

```
// cs1941.cs
using System.Collections;
using System.Linq;
class Test
{
    static int Main()
    {
        var nums = new[] { 1, 2, 3, 4, 5, 6 };
        var words = new string[] { "lake", "mountain", "sky" };
        IEnumerable e = from n in nums
                        join w in words on n equals w // CS1941
                        select w;
        return 0;
    }
}
```

The method in which type inference fails is the method that the query clause is translated to at compile time.

See also

- LINQ in C#
- Type Relationships in LINQ Query Operations

Compiler Error CS1942

Article • 09/21/2022

The type of the expression in the 'clause' clause is incorrect. Type inference failed in the call to 'method'.

This error is typically generated when the range variable has been given an incorrect explicit type.

To correct this error

1. If the range variable is explicitly typed, make sure that the type is either the same as, or implicitly convertible from, the type of the elements in the collection it iterates. If the range variable is preceded with the `var` keyword, remove `var`.

Example

The following code generates CS1942:

C#

```
// cs1942.cs
class Program
    {
        static void Main(string[] args)
        {
            var x = from var i in Enumerable.Range(1, 100) // CS1949
                    select i; //CS1942
        }
    }
```

CS1942 is related to CS1949 because the use of `var` with a range variable causes the underlying `Cast<T>` operation to fail because `var` is not a type.

See also

- var
- LINQ in C#

Compiler Error CS1943

Article • 09/15/2021

An expression of type 'type' is not allowed in a subsequent from clause in a query expression with source type 'type'. Type inference failed in the call to 'method'.

All range variables must represent queryable types.

To correct this error

1. Make sure that the type is a queryable type that implements `IEnumerable`, `IEnumerable<T>` or a derived interface, or any other type which has a query pattern defined for it.

2. If the type is a non-generic `IEnumerable`, provide an explicit type on the range variable.

Example

The following code generates CS1943:

C#

```
// cs1943.cs
using System.Linq;
class Test
{
    class TestClass
    { }
    static void Main()
    {
        int[] nums = { 0, 1, 2, 3, 4, 5 };
        TestClass tc = new TestClass();

        var x = from n in nums
                from s in tc // CS1943
                select n + s;
    }
}
```

Compiler Error CS1947

Article • 09/15/2021

Range variable 'variable name' cannot be assigned to -- it is read only.

A range variable is like an iteration variable in a `foreach` statement. It cannot be assigned to in a query expression.

To correct this error

1. Remove the assignment to the range variable.
2. If necessary, introduce a new range variable by using the `let` clause and use it to store the value.

Example

The following code generates CS1947:

C#

```
// cs1947.cs
using System.Linq;
class Test
{
    static void Main()
    {
        int[] array = new int[] { 1, 2, 3, 4, 5 };
        var x = from i in array
                let k = i
                select i = 5; // CS1947
        x.ToList();
    }
}
```

See also

- LINQ Query Expressions

Compiler Error CS1948

Article • 09/15/2021

The range variable 'name' cannot have the same name as a method type parameter

The same declaration space cannot contain two declarations of the same identifier.

To correct this error

1. Change the name of the range variable or the type parameter.

Example

The following example generates CS1948 because the identifier `T` is used for the range variable and for the type parameter on method `TestMethod`:

C#

```
// cs1948.cs
using System.Linq;
class Test
{
    public void TestMethod<T>(T t)
    {
        var x = from T in Enumerable.Range(1, 100) // CS1948
                select T;
    }
}
```

Compiler Error CS1949

Article • 09/21/2022

The contextual keyword 'var' cannot be used in a range variable declaration.

A range variable is implicitly typed by the compiler. There is no need to use var with a range variable.

To correct this error

1. Remove the `var` keyword from in front of the range variable.

Example

The following example generates CS1949:

C#

```
// cs1949.cs
using System;
using System.Linq;
class Test
{
    static void Main()
    {
        var x = from var i in Enumerable.Range(1, 100) // CS1949
        select i;
    }
}
```

See also

- LINQ Query Expressions
- Introduction to LINQ Queries (C#)

Compiler Error CS1955

Article • 09/15/2021

Non-invocable member 'name' cannot be used like a method.

Only methods and delegates can be invoked. This error is generated when you try to use empty parentheses to call something other than a method or delegate.

To correct this error

1. Remove the parentheses from the expression.

Example

The following code generates CS1955 because the code is trying to invoke a field and a property by using the invocation expression (). You cannot call a field or a property. Use the member access expression . to access the value it stores.

C#

```
// cs1955.cs
class A
{
    public int x = 0;
    public int X
    {
        get { return x; }
        set { x = value; }
    }
}

class Test
{
    static int Main()
    {
        A a = new A();
        a.x(); // CS1955
        a.X(); // CS1955
        // Try this line instead:
        // int num = a.x;
    }
}
```

Compiler Error CS1958

Article • 09/08/2022

Object and collection initializer expressions may not be applied to a delegate creation expression.

A delegate has no members like a class or struct has, and so there is nothing for an object initializer to initialize. If you encounter this error, it is probably because there are braces after the delegate creation expression. Just remove the braces and this error will disappear.

To correct this error

1. Remove the braces.

Example

The following code produces CS1958:

C#

```
// cs1958.cs
public class MemberInitializerTest
{
    delegate void D<T>();
    public static void GenericMethod<T>() { }
    public static void Run()
    {
        D<int> genD = new D<int>(GenericMethod<int>) { }; // CS1958
       // Try the following line instead
      // D<int> genD = new D<int>(GenericMethod<int>);
    }
}
```

Compiler Error CS1959

Article • 09/15/2021

'name' is of type 'type'. The type specified in a constant declaration must be sbyte, byte, short, ushort, int, uint, long, ulong, char, float, double, decimal, bool, string, an enum-type, or a reference-type.

The types permitted in a const declaration are limited to those described in this message.

To correct this error

1. Declare the constant with an allowed type.

Example

The following code produces CS1959 because `null` is not a type.

C#

```
// cs1959.cs
class Program
    {
        static void Test<T>() where T : class
        {
            const T x = null; // CS1959
        }
    }
```

See also

- Constants
- null

Compiler Error CS2001

Article • 09/15/2021

Source file 'file' could not be found

A source file name was passed to the compiler, but could not be located. Check the spelling of the file name and the location of the file.

Compiler Error CS2003

Article • 09/15/2021

Response file 'file' included multiple times

A response file was passed to the compiler more than once. A response file can only be passed to the compiler once per output file.

For more on response files, see **ResponseFiles** (Specify Response File).

Compiler Error CS2005

Article • 09/15/2021

Missing file specification for 'option' option

A compiler option was only partially specified.

For example, when using `-recurse`, you must specify the file to search: **/recurse:*filename*.cs**.

Example

The following sample generates CS2005.

C#

```
// CS2005.cs
// compile with: /recurse:
// CS2005 expected
class x
{
   public static void Main() {}
}
```

Compiler Error CS2006

Article • 09/15/2021

Command-line syntax error: Missing 'text' for 'option' option

The syntax for *option* requires additional text. For information, see Compiler Options.

Compiler Error CS2007

Article • 09/15/2021

Unrecognized command-line option: 'option'

The compiler was passed a string that was not a compiler option, even though it began with a forward slash (/).

The following sample generates CS2007:

C#

```
// CS2007.cs
// compile with: /recur
// CS2007 expected
class x
{
   public static void Main() {}
}
```

Compiler Error CS2008

Article • 09/15/2021

No inputs specified

The compiler was invoked and compiler options were specified, but no source-code files were passed.

Compiler Error CS2011

Article • 09/15/2021

Unable to open response file 'file'

A response file was specified in a compilation, but the compiler was unable to locate and open the file.

For more on response files, see **ResponseFiles** (Specify Response File).

Compiler Error CS2012

Article • 02/17/2024

Cannot open 'file' for writing

While using the `-bugreport:`, `file` compiler option, the file could not be opened for writing. Make sure you specified a valid file name and that the file is not read-only. One possible cause of this error is that the file is used in another process, for example a debugger or an antivirus program.

> **ⓘ Important**
>
> The `bugreport` option is no longer supported.

Compiler Error CS2013

Article • 09/15/2021

Invalid image base number 'value'

An invalid value (not a number) was passed to the **BaseAddress** compiler option.

The following sample generates CS2013:

C#

```
// CS2013.cs
// compile with: /target:library /baseaddress:x
// CS2013 expected
class MyClass
{
}
```

Compiler Error CS2015

Article • 09/15/2021

'file' is a binary file instead of a text file

A file was passed to the compiler that was a binary file. The compiler expects a source code file.

Compiler Error CS2016

Article • 09/15/2021

Code page 'codepage' is invalid or not installed

The **CodePage** compiler option was passed an invalid value.

The following sample generates CS2016:

C#

```
// CS2016.cs
// compile with: /codepage:x
// CS2016 expected
class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS2017

08/08/2025

Cannot specify /main if building a module or library

You cannot specify a main entry point when you are building a **library** OutputType.

The following sample generates CS2017:

C#

```
// CS2017.cs
// compile with: /main:MyClass /target:library
// CS2017 expected
class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Error CS2018

Article • 09/15/2021

Unable to find messages file 'cscmsgs.dll'

The .dll file that contains the compiler's error and warning messages was not found. This file must be present in the same directory as the other compiler support files.

Compiler Error CS2019

08/08/2025

Invalid target type for /target: must specify 'exe', 'winexe', 'library', or 'module'

The **OutputType** compiler option was used, but an invalid parameter was passed. To resolve this error, recompile the program using the form of the **/target** option that is appropriate to your output file.

The following sample generates CS2017:

C#

```
// CS2019.cs
// compile with: /target:libra
// CS2019 expected
class MyClass
{
}
```

Compiler Error CS2020

Article • 09/15/2021

Only the first set of input files can build a target other than 'module'

In a multi-output compilation, the first output file must be built with -target:exe, -target:winexe, or -target:library. Any subsequent output files must be built with -target:module.

Compiler Error CS2021

Article • 09/08/2022

File name 'file' is too long or invalid

All file names passed to the C# compiler must be no longer than `_MAX_PATH` (defined in a Windows header file). The compiler will give this error in the following situations:

- A file name (including the path) is longer than `_MAX_PATH`.
- The file name contains invalid characters.
- The file name contains wildcards where wildcards are not allowed (such as in resource file names).

Compiler Error CS2022

Article • 03/18/2022

Options '/out' and '/target' must appear before source file names

The **-out** (Set Output Filename) and -target (Specify Output File Format) compiler options, when specified on the command line, must precede the source code files.

Compiler Error CS2024

Article • 09/15/2021

Invalid file section alignment number '#'

An invalid value was passed to the **FileAlignment** compiler option.

Example

The following sample generates CS2024:

C#

```
// CS2024.cs
// compile with: /filealign:ex
// CS2024 expected
class MyClass
{
    public static void Main()
    {
    }
}
```

Compiler Error CS2032

Article • 09/15/2021

Character 'character' is not allowed on the command-line or in response files

Response files and the command line options for csc.exe cannot contain ASCII control characters in the range 0-31 or the pipe (`|`) character.

Compiler error CS2032 is difficult to demonstrate from the command line because the command line processor and the integrated development environment (IDE) filter out characters that are not valid. The following procedure generates the error by using a response file.

To generate this error

1. In the *My Documents* folder, create a text file that is named *CS2032.rsp*, and then enter the following compiler options in it:

   ```
   Console

   /target:exe /out:cs|2032.exe cs2032.cs
   ```

2. In the *My Documents* folder, create a text file that's named *cs2032.cs* and that contains whatever you want.

3. Open Visual Studio Developer Command Prompt or Visual Studio Developer PowerShell.

4. Change the current directory to `C:\Users\\<YourUsername>\Documents`.

5. Run the following command from the command prompt: `csc @cs2032.rsp`

6. The CS2032 error message appears because the response file, *CS2032.rsp*, contains a pipe character.

Compiler Error CS2033

Article • 09/15/2021

Cannot create short filename 'filename' when a long filename with the same short filename already exists

Compile any C# file with a name longer than eight characters. Then compile another file with the short version of the preceding file name, such as the first six characters of the name plus "~1." The second compile will generate this error.

To resolve this error, rename the short file name to one that does not conflict with the long file name.

Compiler Error CS2034

Article • 09/15/2021

A /reference option that declares an extern alias can only have one filename. To specify multiple aliases or filenames, use multiple /reference options.

To specify two aliases and/or file names, use two **/reference** options, like this:

Example

The following code will generate error CS2034.

C#

```
// CS2034.cs
// compile with: /r:A1=cs2034a1.dll;A2=cs2034a2.dll
// to fix, compile with: /r:A1=cs2034a1.dll /r:A2=cs2034a2.dll
// CS2034
extern alias A1;
extern alias A2;
using System;
```

Compiler Error CS2035

Article • 09/15/2021

Command-line syntax error: Missing ':<number>' for 'compiler_option' option

Some compiler options require a value.

Example

The following sample generates CS2035.

C#

```
// CS2035.cs
// compile with: /baseaddress
// CS2035 expected
```

Compiler Error CS2036

Article • 09/15/2021

The /pdb option requires that the /debug option also be used.

Program database files are only generated for debug builds. The **/pdb** option is therefore meaningless in a retail build.

To correct this error

- Add the **/debug** compiler option.
- Remove the **/pdb** compiler option.

Example

The following example generates CS2036 when it is compiled with the **/pdb** option but not the /debug option:

C#

```
// cs2036.cs
// Compile with: /pdb
// CS2036
class Test
{
    public static int Main()
    {
        return 1;
    }
}
```

See also

- **PdbFile** (C# Compiler Options)
- **DebugType** (C# Compiler Options)

Compiler Error CS5001

07/22/2025

Program does not contain a static 'Main' method suitable for an entry point

This error occurs when no static `Main` method with a correct signature is found in the code that produces an executable file. It also occurs if the entry point function, `Main`, is defined with the wrong case, such as lower-case `main`. For information about the rules that apply to the `Main` method, see Main() and Command-Line Arguments.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

If the `Main` method has an `async` modifier, make sure that the selected C# language version is 7.1 or higher and to use `Task` or `Task<int>` as the return type.

The `Main` method is only required when compiling an executable file, that is, when the **exe** or **winexe** element of the **TargetType** compiler option is specified. The following Visual Studio project types specify one of these options by default:

- Console application
- ASP.NET Core application
- WPF application
- Windows Forms application

Example

The following example generates CS5001:

C#

```
// CS5001.cs
// CS5001 expected when compiled with -target:exe or -target:winexe
public class Program
{
   // Uncomment the following line to resolve.
   // static void Main() {}
}
```

Compiler Error CS7003

Article • 09/23/2023

Unexpected use of an unbound generic name

This error occurs if you use a generic type needing one parameter type without passing any generic parameter type name between the angle brackets. This use may be a variable declaration, or an object instantiation.

Example

The following example generates CS7003:

C#

```
// CS7003.cs
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        var myDictionary = new Dictionary< , >();  //CS7003

        List<> var2;                               //CS7003
    }
}
```

To correct this error

Provide the expected parameter type names in angle brackets, separated by commas, when using a generic type.

The previous example could be fixed as follows :

C#

```
// CS7003-fixed.cs
using System.Collections.Generic;

class Program
{
    static void Main(string[] args)
    {
        var myDictionary = new Dictionary<int, string>();
```

```
        List<string> var2;
    }
}
```

See also

- Generics

Compiler Error CS8070

09/09/2025

Control cannot fall out of switch from final case label ('label')

This error occurs when the last `case` or `default` case of a switch statement doesn't end with a jump statement such as:

- break
- return
- throw

The following sample generates CS8070:

C#

```
// CS8070.cs
public class MyClass
{
    public static void Main()
    {
        int i = 2;
        int j = 0;

        switch (i)
        {
            case 1:
                i++;
                break;

            // Compiler error CS8070 is reported on the following line.
            case 2:
                i += 2;
            // To resolve the error, uncomment one of the following example
statements.
            // break;
            // return;
            // throw new Exception("Fin");
        }

        switch (j)
        {
            case 1:
                j++;
                break;

            case 2:
                j += 2;
                break;
```

```
            // Compiler error CS8070 is reported on the following line.
            default:
                Console.WriteLine("Default");
            // To resolve the error, uncomment the following line:
            // break;
        }
    }
}
```

Compiler Error CS8124

Article • 10/04/2022

Tuple must contain at least two elements.

Example

The following sample generates CS8124:

```
C#
```

```
// CS8124.cs (4,20)

class C
{
    void M(int x, () y, (int a) z) { }
}
```

To correct this error

A valid tuple declaration contains two or more elements, ensuring tuple declarations have two or more elements corrects this error:

```
C#
```

```
class C
{
    void M(int x, (int, int) y, (int a, int b) z) { }
}
```

Compiler Error CS8125

Article • 10/04/2022

Tuple element name is only allowed at position.

Example

The following sample generates CS8125:

```
C#
```

```
// CS8125.cs (2,15)

public class C
{
    public void Method()
    {
        var tuple3 = (Item2: 2, Item1: 1);
    }
}
```

To correct this error

If tuple element names `Item1`, `Item2`, etc. are used, ensuring the correct order corrects this error:

```
C#
```

```
    public void Method()
    {
        var tuple3 = (Item1: 2, Item2: 1);
    }
```

Compiler Error CS8127

Article • 10/04/2022

Tuple element names must be unique.

Example

The following sample generates CS8127:

C#

```
// CS8127.cs (7,0)

internal struct NewStruct
{
    public int a;
    public int b;

    public static implicit operator (int a, int a)(NewStruct value)
    {
        return (value.a, value.b);
    }
}
```

To correct this error

Ensuring the names of elements within a tuple declaration are unique corrects this error:

C#

```
    public static implicit operator (int a, int b)(NewStruct value)
    {
        return (value.a, value.b);
    }
```

Compiler Error CS8129

Article • 10/04/2022

No suitable 'Deconstruct' instance or extension method was found for type, with `out` parameters and a `void` return type.

Example

The following sample generates CS8129:

C#

```
// CS8129.cs (11,52)

class C
{
    static void Main()
    {
        long x;
        string y;
        (x, y) = new C();
    }

    public int Deconstruct(out int a, out string b)
    {
        a = 1;
        b = "hello";
        return 42;
    }
}
```

To correct this error

A valid `Deconstruct` method returns `void` and has two or more `out` parameters that match the type of a tuple that would be deconstructed. Implementing a valid `Deconstruct` method corrects this error:

C#

```
    public void Deconstruct(out int a, out string b)
    {
        a = 1;
        b = "hello";
    }
```

Compiler Error CS8130

Article • 10/07/2022

Cannot infer the type of implicitly-typed deconstruction variable.

Example

The following sample generates CS8130:

```
C#
```

```
// CS8130.cs (5,14)
class Program
{
    static void Main()
    {
        var (x2, y2) = () => { };
    }
}
```

The compiler cannot convert a delegate (`Action`) to a two-element tuple and thus is unable to infer the type of each element of the tuple.

To correct this error

To assign a value to a tuple, ensuring the right-hand-side expression is a tuple with the same number of elements as that on the left-hand-side corrects this error:

```
C#
```

```
    static void Main()
    {
        var (x2, y2) = (1, 2);
    }
```

Compiler Error CS8131

Article • 10/07/2022

Deconstruct assignment requires an expression with a type on the right-hand-side.

Example

The following sample generates CS8131:

```
C#
```

```
// CS8131.cs (5,24)
class Program
{
    static void Main()
    {
        var (x2, y2) = () => { };
    }
}
```

The compiler cannot convert a delegate (`Action`) to a two-element tuple, resulting in CS8131.

To correct this error

To assign a value to a tuple, ensure the right-hand-side expression is the same type of tuple as that on the left-hand-side:

```
C#
```

```
    static void Main()
    {
        var (x2, y2) = (1, 2);
    }
```

Compiler Error CS8132

Article • 10/04/2022

Cannot deconstruct a tuple of elements into variables.

Example

The following sample generates CS8132:

C#

```
// CS8132.cs (5,9)
class Program
{
    static void F(object x, object y, object? z)
    {
        (object? a, object? b) = (x, y, z = 3);
        Console.WriteLine(a);
        Console.WriteLine(b);
        Console.WriteLine(z);
    }
}
```

To correct this error

To deconstruct to a tuple with a smaller number of elements, using the discard variables will correct this error:

C#

```
class Program
{
    static void F(object x, object y, object? z)
    {
        (object? a, object? b, object _) = (x, y, z = 3);
        Console.WriteLine(a);
        Console.WriteLine(b);
        Console.WriteLine(z);
    }
}
```

Compiler Error CS8139

Article • 10/04/2022

cannot change tuple element names when overriding inherited member

Example

The following sample generates CS8139:

```
C#
```

```
// CS8139.cs (9,38)

public class Base
{
    public virtual (object a, object b) M((object c, object d) x) { return
x; }
}

class C : Base
{
    public override (object, object) M((object c, object d) y) { return y; }
}
```

To correct this error

Ensuring the tuple element names in the overriding member match those in the virtual member corrects this error:

```
C#
```

```
class C : Base
{
    public override (object a, object b) M((object c, object d) y) { return
y; }
}
```

Compiler Error CS8140

Article • 10/04/2022

is already listed in the interface list on type with different tuple element names, as.

Example

The following sample generates CS8140:

C#

```
// CS8140.cs (11,7)

interface I<T>
{
    T GetValue();
}

interface I2 : I<(int c, int d)>
{
}

class C : I<(int a, int b)>, I2
{
    public (int c, int d) GetValue()
    {
        return (1, 2);
    }
}
```

To correct this error

Renaming the tuple element names to match the interface declaration corrects this error:

C#

```
interface I<T>
{
    T GetValue();
}

interface I2 : I<(int c, int d)>
{
}
```

```
class C : I<(int c, int d)>, I2
{
    public (int c, int d) GetValue()
    {
        return (1, 2);
    }
}
```

Compiler Error CS8141

Article • 11/03/2022

The tuple element names in the signature of method must match the tuple element names of interface method (including on the return type).

Example

The following sample generates CS8141:

```
C#
```

```
// CS8141.cs (10,27)
using System.Collections;

public interface IGrabber<out T>
{
    T GetOne();
}

class SomeGrabber : IGrabber<(int, int)>
{
    public (int a, int b) GetOne()
    {
        return (1, 2);
    }
}
```

To correct this error

Changing the signature of the `GetOne` method to return an unnamed tuple, matching the unnamed tuple in the interface, will correct this error:

```
C#
```

```
    public (int, int) GetOne()
    {
        return (1, 2);
    }
```

Compiler Error CS8145

Article • 10/26/2024

Auto-implemented properties cannot return by reference

Automatically implemented properties are not guaranteed to have a member or variable that can be referenced and thus do not support return by reference.

Example

The following sample generates CS8145:

```
C#
```

```
// CS8145.cs (4,13)

public class C
{
    public ref int Property1 { get; }
}
```

To correct this error

If the property can be implemented through a backing field, then refactoring to use a backing field and `ref`-returning the field will correct this error:

```
C#
```

```
public class C
{
    private int property1;

    public ref int Property1 => ref property1;
}
```

If the property cannot be implemented through a backing field, then removing the `ref` modifier from the property corrects this error:

```
C#
```

```
public class C
{
```

```
    public int Property1 { get; }
}
```

Compiler Error CS8146

Article • 09/27/2022

Properties which return by reference must have a get accessor

Example

The following sample generates CS8146:

```
C#
```

```
// CS8146.cs (5,18)

public class C
{
    private ref int number;

    private ref int Number { init => number = value; }
}
```

To correct this error

Ensuring a property that returns by reference has a get accessor will correct this error:

```
C#
```

```
public class C
{
    private ref int number;

    private ref int Number => ref number;
}
```

Compiler Error CS8147

Article • 10/07/2022

Properties which return by reference cannot have set accessors

Example

The following sample generates CS8147:

```
C#
```

```
// CS8147.cs (6,44)

public class C
{
    private ref int number;

    ref int Number { get => ref number; init => number = value; }
}
```

To correct this error

Removing any set accessor for a property that returns by reference will correct this error:

```
C#
```

```
public class C
{
    private ref int number;

    ref int Number => ref number;
}
```

Compiler Error CS8148

Article • 09/27/2022

must match by reference return of overridden member

Example

The following sample generates CS8148:

```
C#
```

```
// CS8148.cs (11,29)

public class Base
{
    public virtual int GetNumber() { return 0; }
}

public class Derived : Base
{
    private int number;

    public override ref int GetNumber() { return ref number; }
}
```

To correct this error

Ensuring that base members that return by value are implemented by members that do not return by reference corrects this error:

```
C#
```

```
public class Base
{
    public virtual int GetNumber() { return 0; }
}

public class Derived : Base
{
    private int number;

    public override int GetNumber() { return number; }
}
```

Compiler Error CS8149

Article • 09/27/2022

By-reference returns may only be used in methods that return by reference

Example

The following sample generates CS8149:

```
C#
```

```
// CS8149.cs (9,33)

delegate int E();

class C
{
    static int i;
    static void M()
    {
        var e = new E(() => ref i);
    }
}
```

To correct this error

Ensuring that methods and delegates declared as a returning by value do not return by reference corrects this error:

```
C#
```

```
delegate int E();

class C
{
    static int i;
    static void M()
    {
        var e = new E(() => i);
    }
}
```

Compiler Error CS8150

Article • 09/24/2022

By-value returns may only be used in methods that return by value

Example

The following sample generates CS8150:

```
C#
```

```
// CS8150.cs (6,9)

class C
{
    ref int M()
    {
        return new();
    }
}
```

To correct this error

To return by value, refactor the method from being by reference. For example:

```
C#
```

```
// CS8150.cs (6,9)

class C
{
    int M()
    {
        return new();
    }
}
```

Compiler Error CS8151

Article • 09/27/2022

The return expression must be of type because this method returns by reference

Example

The following sample generates CS8151:

```
C#
```

```
// CS8151.cs (6,20)

class Program
{
    ref int M(ref long i)
    {
        return ref i;
    }
}
```

To correct this error

Ensuring that a `ref` return matches the type of the referenced variable regardless of any implicit conversions corrects this error:

```
C#
```

```
class Program
{
    ref long M(ref long i)
    {
        return ref i;
    }
}
```

Compiler Error CS8152

Article • 09/24/2022

Type does not implement interface member. cannot implement because it does not have matching return by reference.

Example

The following sample generates CS8152:

To implement an interface with a method that returns by reference, the implementation of the method must also return by reference and not by value.

C#

```
// CS8152.cs (6,21)

public interface ITest
{
    ref readonly int M();
}
public class Test : ITest
{
    public int M() => 0;
}
```

To correct this error

Ensure interface methods that return by reference do not return by value. For example:

C#

```
public class Test : ITest
{
    int m;
    public ref readonly int M() => ref m;
}
```

Compiler Error CS8156

Article • 09/24/2022

An expression cannot be used in this context because it may not be passed or returned by reference

Example

The following sample generates CS8156:

C#

```
// CS8156.cs (7,27)

class Test
{
    delegate ref int D1();

    void Test1()
    {
        D1 d1 = () => ref 2 + 2;
    }
}
```

To correct this error

When not using referenceable variables, refactoring to return by value corrects this error:

C#

```
class Test
{
    delegate int D1();

    void Test1()
    {
        D1 d1 = () => 2 + 2;
    }
}
```

Compiler Error CS8157

Article • 09/24/2022

Cannot return by reference because it was initialized to a value that cannot be returned by reference

Example

The following sample generates CS8157:

C#

```
// CS8157.cs (8,21)

class C
{
    ref int M()
    {
        int x = 0;
        ref int rx = ref x;
        return ref (rx = ref (new int[1])[0]);
    }
}
```

To correct this error

To return a value that cannot be returned by reference, refactoring to return by value corrects this error:

C#

```
class C
{
    int M()
    {
        int x = 0;
        ref int rx = ref x;
        return rx = ref (new int[1])[0];
    }
}
```

Compiler Error CS8158

Article • 09/24/2022

Cannot return by reference a member because it was initialized to a value that cannot be returned by reference

Example

The following sample generates CS8158:

C#

```
// CS8158.cs (11,14)

public class Test
{
    public struct S1
    {
        public char x;
    }

    ref char Test1(char arg1, S1 arg2)
    {
        ref S1 r = ref arg2;
        return ref r.x;
    }
}
```

To correct this error

To return members initialized to a value that cannot be returned by reference, refactoring to return by value corrects this error:

C#

```
    char Test1(char arg1, S1 arg2)
    {
        ref S1 r = ref arg2;
        return r.x;
    }
```

Compiler Error CS8159

Article • 09/24/2022

Cannot return the range variable by reference

Example

The following sample generates CS8159:

C#

```
// CS8159.cs (7,74)

using System.Linq;
class TestClass
{
    delegate ref char RefCharDelegate();
    void TestMethod()
    {
        var x = from c in "TestValue" select (RefCharDelegate)(() => ref c);
    }
}
```

To correct this error

To return a range variable, refactoring to return by value correct this error:

C#

```
using System.Linq;
class TestClass
{
    delegate char RefCharDelegate();
    void TestMethod()
    {
        var x = from c in "TestValue" select (RefCharDelegate)(() => c);
    }
}
```

Compiler Error CS8160

Article • 09/30/2023

A readonly field cannot be returned by writable reference

Example

The following sample generates CS8160:

```
C#
```

```
// CS8160.cs (8,20)

class Program
{
    readonly int i = 0;

    ref int M()
    {
        return ref i;
    }
}
```

To correct this error

To return a `readonly` field, refactor to return by value:

```
C#
```

```
class Program
{
    readonly int i = 0;

    int M()
    {
        return i;
    }
}
```

Compiler Error CS8161

Article • 09/24/2022

A static readonly field cannot be returned by writable reference

Example

The following sample generates CS8161:

```
C#
```

```
// CS8161.cs (12,14)
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly char s1;

    ref char Test2()
    {
        return ref s1;
    }
}
```

To correct this error

To return the value of a `static readonly` field, refactor to return by value:

```
C#
```

```
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly char s1;

    char Test2()
    {
        return s1;
```

```
    }
}
```

Compiler Error CS8162

Article • 09/24/2022

Members of readonly field cannot be returned by writable reference

Example

The following sample generates CS8162:

```
C#
```

```
// CS8162.cs (12,14)
public class Test
{
    public struct S1
    {
        public char x;
    }

    public readonly S1 i2;

    ref char Test1()
    {
        return ref i2.x;
    }
}
```

To correct this error

To return the value of a `readonly` field, refactoring to return by value corrects this error:

```
C#
```

```
public class Test
{
    public struct S1
    {
        public char x;
    }

    public readonly S1 i2;
    char Test1()
    {
        return i2.x;
    }
}
```

Compiler Error CS8163

Article • 09/24/2022

Fields of static readonly field cannot be returned by writable reference

Example

The following sample generates CS8163:

C#

```
// CS8163.cs (12,14)
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly S1 s2;

    char Test2()
    {
        return s2.x;
    }
}
```

To correct this error

To return the value of a `static readonly` field, refactoring to return by value corrects this error:

C#

```
public class Test
{
    public struct S1
    {
        public char x;
    }

    public static readonly S1 s2;

    char Test2()
    {
        return s2.x;
```

```
    }
}
```

Compiler Error CS8170

07/08/2025

Struct members cannot return 'this' or other instance members by reference

Example

The following sample generates CS8170:

Value types (i.e. `struct`s), are most commonly allocated on the stack. A value type allocated on the stack becomes invalid leaving the scope in which it was declared. The compiler avoids a reference to a variable that becomes invalid leaving the scope by generating this error.

C#

```
// CS8170.cs (8,14)

struct Program
{
    public int d;

    public ref int M()
    {
        return ref d;
    }
}

public class Other
{
    public void Method()
    {
        var p = new Program();
        ref int d = ref p.M();
    }
}
```

To correct this error

Changing the method to not `ref`-return corrects this error:

C#

```
delegate void D();

struct Program
{
```

```
    public event D d;

    public D M()
    {
        return d;
    }
}
```

If a reference to a member is required, consider extending the scope of the value. For example:

C#

```
public struct Program
{
    public int d;
}

public static class Extensions
{
    public static ref readonly int RefD(this in Program program)
    {
        return ref program.d;
    }
}

public class Other
{
    public void Method()
    {
        var p = new Program();
        var d = p.RefD();
    }
}
```

Another approach is to use the System.Diagnostics.CodeAnalysis.UnscopedRefAttribute attribute. It will mark the reference to be allowed to escape the scope.
Use this only when you know that it is safe for the reference to leave the scope. Below is the example of applying System.Diagnostics.CodeAnalysis.UnscopedRefAttribute to `int M()` method, which fixes the CS8170:

C#

```
using System.Diagnostics.CodeAnalysis;

struct Program
{
    public int d;

    [UnscopedRef]
    public ref int M()
```

```
    {
        return ref d;    // No error - ref is valid to escape the scope in this
line of that method
    }
}

public class Other
{
    public void Method()
    {
        var p = new Program();
        ref int d = ref p.M();
    }
}
```

Compiler Error CS8171

Article • 09/21/2022

Cannot initialize a by-value variable with a reference

Example

The following sample generates CS8171:

C#

```
// CS8171.cs (8,13)

class Test
{
    void A()
    {
        int a = 123;
        ref int x = ref a;
        var y = ref x;
    }
}
```

Remember that `var y = ref x` is implicitly `int y = ref x` where `int y` is a by-value variable.

To correct this error

Removing the `ref` modifier from the right side of the assignment will correct this error:

C#

```
class Test
{
    void A()
    {
        int a = 123;
        ref int x = ref a;
        var y = x;
    }
}
```

Compiler Error CS8172

Article • 09/22/2022

Cannot initialize a by-reference variable with a value

Example

The following sample generates CS8172:

C#

```
// CS8172.cs (10,17)

class C
{
    void M()
    {
        ref readonly int L() => ref (new int[1])[0];

        ref readonly int x = ref L();
        ref int y = x;
    }
}
```

To correct this error

Assigning a reference to a variable to a by-reference variable corrects this error:

C#

```
class C
{
    void M()
    {
        ref readonly int L() => ref (new int[1])[0];

        ref readonly int x = ref L();
        ref int y = ref x;
    }
}
```

Compiler Error CS8173

Article • 09/21/2022

The expression must be of type because it is being assigned by reference

When assigning a reference to a variable, the type of the variables must match to be referenceable.

Example

The following sample generates CS8173:

```
C#
```

```
// CS8173.cs (12,18)

class C
{
    void M()
    {
        string s = "s";
        object o = s;
        ref string rs = ref s;
        ref object ro = ref o;

        ro = ref s;
    }
}
```

To correct this error

Assigning the reference to a variable of the correct type will correct this error:

```
C#
```

```
        rs = ref s;
```

Compiler Error CS8174

Article • 09/21/2022

A declaration of a by-reference variable must have an initializer

Example

The following sample generates CS8174:

```
C#
```

```
// CS8174.cs (7,22)

class C
{
    void M()
    {
        int i = 0;
        for (ref int rx; i < 5; i++) { }
    }
}
```

To correct this error

Initializing the by-reference variable with a reference to a variable will correct this error:

```
C#
```

```
class C
{
    void M()
    {
        int i = 0;
        for (ref int rx = ref i; i < 5; i++) { }
    }
}
```

Compiler Error CS8177

Article • 07/03/2024

Async methods cannot have by-reference locals

To manage asynchronous state, `async` methods use a state machine, capturing variable state in closures implemented in compiler-generated classes and properties. A local variable reference (on the stack) can't be captured within the instance of a class in the heap, so the compiler issues an error.

Example

The following sample generates CS8177 before C# 13:

C#

```
// CS8177.cs (20,26)

using System.Threading.Tasks;

class E
{
    public class Enumerator
    {
        public ref int Current => throw new
System.NotImplementedException();
        public bool MoveNext() => throw new
System.NotImplementedException();
    }

    public Enumerator GetEnumerator() => new Enumerator();
}

class C
{
    public async static Task Test()
    {
        await Task.CompletedTask;

        foreach (ref int x in new E())
        {
            System.Console.Write(x);
        }
    }
}
```

To correct this error

Remove the `ref` modifier. Or, you can upgrade to C# 13, which ships with .NET 9.

C#

```
class C
{
    public async static Task Test()
    {
        await Task.CompletedTask;

        foreach (int x in new E())
        {
            System.Console.Write(x);
        }
    }
}
```

Compiler Error CS8178

07/22/2025

'await' cannot be used in an expression containing a call to because it returns by reference

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

Example

The following sample generates CS8178:

C#

```
using System;
using System.Threading.Tasks;

class TestClass
{
    int x;
    ref int Save(int y)
    {
        x = y;
        return ref x;
    }

    async Task TestMethod()
    {
        Save(1) = await Task.FromResult(0);
    }
}
```

To correct this error

Changing the use of the return by reference to be synchronous corrects the error:

C#

```
    async Task TestMethod()
    {
```

```
        var x = await Task.FromResult(0);
        Save(1) = x;
    }
```

Compiler Error CS8210

Article • 10/28/2022

A tuple may not contain a value of type 'void'.

Example

The following sample generates CS8210:

```C#
// CS8210.cs
void Method()
{
}

void Test()
{
    var x = ("something", Method());
}
```

Your method returns `void`, so that method doesn't return a value. You can't use a method that returns `void` for a data member of a tuple.

See also

- Tuple
- Void

Compiler Error CS8333

Article • 10/01/2022

Cannot return by writable reference because it is a readonly variable

Example

The following sample generates CS8333:

```
C#
```

```
// CS8333.cs (3,36)
class Program
{
    static ref T F4<T>(in T t) => ref t;
}
```

To correct this error

To return a reference to a readonly variable, refactoring to return `ref readonly` will correct this error:

```
C#
```

```
class Program
{
    static ref readonly T F4<T>(in T t) => ref t;
}
```

Compiler Error CS8334

Article • 10/01/2022

Members of cannot be returned by writable reference because it is a readonly variable

Example

The following sample generates CS8334:

```
C#
```

```
// CS8334.cs (5,14)

class Program
{
    static ref int M(in int arg1, in (int Alice, int Bob) arg2)
    {
        return ref arg2.Alice;
    }
}
```

To correct this error

To return a reference to a read-only member, refactoring to return `ref readonly` will correct this error:

```
C#
```

```
class Program
{
    static ref readonly int M(in int arg1, in (int Alice, int Bob) arg2)
    {
        return ref arg2.Alice;
    }
}
```

Compiler Error CS8352

Article • 04/18/2024

Cannot use variable 'variable' in this context because it may expose referenced variables outside of their declaration scope.

This error indicates potential undefined behavior when there's a risk of accessing a stack-allocated variable after it's already been deleted from the stack.

To correct this error

You fix `CS8352` by ensuring the variable's scope is at least as wide as the variable that refers to it. You do this by allocating the storage on a heap, or in the stack on the calling context.

Example

The following sample generates CS8352:

C#

```
class Program
{
    static public Span<int> CreateSpanWithValue(int size, int value)
    {
        Span<int> localSpan = stackalloc int[size];
        localSpan[0] = value;
        return localSpan;    // CS8352
    }                        // localSpan refers to stack memory that's been
reclaimed.

    static public void Main()
    {
        Span<int> localSpan = CreateSpanWithValue(5, 10);
        localSpan[2] = 14;
        foreach(var item in localSpan)
        {
            Console.WriteLine(item.ToString());  // Reading from deleted
object
        }
    }
    // Example of output:
    // 284945320
    // 149
    // 149
    // 1369466325
```

```
    // 284945320
}
```

The following sample moves the stack allocation so its scope matches the `Span`.
This example solution works, because the memory Span refers to is allocated on the stack, but in the calling context:

C#

```
class Program
{
    static public void FillSpanWithValue(Span<int> span, int value)
    {
        span[0] = value;
    }

    static public void Main()
    {
        Span<int> localSpan = stackalloc int[5];
        FillSpanWithValue(localSpan, 10);
        localSpan[2] = 14;
        foreach(var item in localSpan)
        {
            Console.WriteLine(item.ToString());
        }
    }
    // Output:
    // 10
    // 0
    // 14
    // 0
    // 0
}
```

The following sample uses the `new` keyword to create the object on heap, instead of stack.
This example solution works, because after returning from `CreateSpanWithValue` method the memory that Span refers to is still valid as it is allocated on the heap, not on a stack:

C#

```
class Program
{
    static public Span<int> CreateSpanWithValue(int size, int value)
    {
        Span<int> localSpan = new int[size];
        localSpan[1] = value;
        return localSpan;
    }
```

```
    static public void Main()
    {
        var span = CreateSpanWithValue(5, 10);
        span[2] = 14;
        foreach(var item in span)
        {
            Console.WriteLine(item.ToString());
        }
    }
    // Output:
    //0
    //10
    //14
    //0
    //0
}
```

See also

- in parameter modifier
- new operator
- stackalloc expression

Compiler Error CS8354

Article • 10/01/2022

Cannot return 'this' by reference.

A `ref` argument must be an assignable variable or array element. `this` is not assignable and cannot be returned by reference.

Example

The following sample generates CS8354:

C#

```
// CS8354.cs (6,20)

class Program
{
    ref Program M()
    {
        return ref this;
    }
}
```

Compiler Error CS8355

Article • 12/13/2023

An in parameter cannot have the Out attribute.

The parameter of a method can't be declared as read only, while informing interop marshaler to treat this parameter as writable by the native code.
OutAttribute will contradict to the `in` keyword when marshalling.

Example

The following sample generates CS8355:

```
C#
```

```
using System.Runtime.InteropServices;

class Program
{
    public static void Example([Out] in int value)    // CS8355
    {
    }
}
```

See also

- Default Marshalling Behavior
- Reference parameters

Compiler Error CS8403

Article • 09/15/2021

Method 'method' with an iterator block must be 'async' to return '{IAsyncEnumerable<T>|IAsyncEnumerator<T>}'

To correct this error

Mark your method with the `async` modifier.

See also

- async (C# Reference)

Compiler Error CS8411

Article • 09/15/2021

Asynchronous foreach statement cannot operate on variables of type 'type' because 'type' does not contain a suitable public instance definition for 'GetAsyncEnumerator'

`await foreach` statement operates only on types having a definition of `GetAsyncEnumerator`, such as `IAsyncEnumerable<T>`.

To correct this error

Replace `await foreach` with `foreach`.

Example

C#

```
using System.Collections.Generic;
using System.Threading.Tasks;

class Program
{
    async Task Example(IAsyncEnumerator<int> enumerator)
    {
        // error CS8411: Asynchronous foreach statement cannot operate on
variables
        // of type 'IAsyncEnumerator<T>' because 'IAsyncEnumerator<T>' does
not
        // contain a suitable public instance definition for
'GetAsyncEnumerator'
        await foreach (int i in enumerator)
        {
        }
    }
}
```

Compiler Error CS8422

Article • 12/19/2023

A static local function cannot contain a reference to 'this' or 'base'.

The static keyword on a local function prevents the local function from accessing the state of parent instance methods or instances fields.
This error indicates the local function accesses instance members of the containing type.

Example

The following sample generates CS8422:

C#

```
public class C
{
    private int counter = 1;
    public void IncreaseCounter()
    {
        static void localFunc(int addition)
        {
            this.counter += addition;   // CS8422 due to reference to 'this'
            base.ToString();            // CS8422 due to reference to 'base'

            // Also for implicit 'this' or 'base' references:
            counter += addition;   // CS8422 due to implicit reference to
'this'
            ToString();            // CS8422 due to implicit reference to
'base'
        }
        localFunc(10);
        Console.WriteLine(this.counter);
    }
}
```

To correct this error

If there is an intention of local function to capture and modify the state of a parent then it shouldn't be declared static. You could also remove access to any instance members.

C#

```
public class C
{
```

```
    private int counter = 1;
    public void IncreaseCounter()
    {
        void localFunc(int addition)
        {
            counter += addition;
            ToString();
        }
        localFunc(10);
        Console.WriteLine(this.counter);
    }
}

public class Program
{
    public void Main()
    {
        C c = new();
        c.IncreaseCounter();
    }
}

// Output:
// 11
```

See also

- Local functions
- CS0120

Compiler Error CS8515

Article • 10/27/2022

Parentheses are required around the switch governing expression.

Example

The following code snippets generate CS8515:

Missing parentheses: `(` `)`:

```
C#

// CS8515.cs
int x = 5;
switch x
{
    case 5:
    break;
}
```

Missing curly brackets: `{` `}`:

```
C#

// CS8515.cs
int x = 5;
switch (x)
    case 5:
    break;
```

The correct format

```
C#

int x = 5;
switch (x)
{
    case 5:
    break;
}
```

For more information, see Switch.

Compiler Error CS8802

Article • 11/27/2024

Only one compilation unit can have top-level statements.

This error indicates that there are two or more top-level statements in a single compilation unit (single project or a single group of files compiled into a single binary file).

Example

The following sample of single compilation unit generates CS8802:

XML

```xml
<!-- SingleCompilationUnit.csproj -->
<Project Sdk="Microsoft.NET.Sdk">
  <PropertyGroup>
    <TargetFramework>net8.0</TargetFramework>
  </PropertyGroup>
</Project>
```

C#

```
// EntryFile.cs

int a = 0;
```

C#

```
// SecondaryEntryFile.cs

int b = 1;    // CS8802: The top level statement already exists in
EntryFile.cs
```

To correct this error

Use only one top-level statement in the project.
Top-level statements acts as an entry point to the program, so only one file may have top-level statement. All other statements must be defined as members of classes or structs.

See also

- Top-level statements
- CS8803

Compiler Error CS8803

Article • 09/17/2022

Top-level statements must precede namespace and type declarations.

Example

The following sample generates CS8803:

```
C#
```

```
// CS8803.cs (0,0)

public record Person
{
    public string? GivenName { get; set; }
    public string? FamilyName { get; set; }
}

int i = 0;
```

In a file with top-level statements, top-level statements must occur prior to any type declarations.

To correct this error

Move the code before the namespace declaration:

```
C#
```

```


int i = 0;

public record Person
{
    public string? GivenName { get; set; }
    public string? FamilyName { get; set; }
}
```

It is common that types are declared within their own file, which would also correct this error by separating the type declaration from the top-level statements.

Compiler Error CS8515

Article • 10/27/2022

Parentheses are required around the switch governing expression.

Example

The following code snippets generate CS8515:

Missing parentheses: `(` `)`:

```
C#

// CS8515.cs
int x = 5;
switch x
{
    case 5:
    break;
}
```

Missing curly brackets: `{` `}`:

```
C#

// CS8515.cs
int x = 5;
switch (x)
    case 5:
    break;
```

The correct format

```
C#

int x = 5;
switch (x)
{
    case 5:
    break;
}
```

For more information, see Switch.

Compiler Error CS9036

Article • 05/30/2025

Required member 'memberName' must be assigned a value, it cannot use a nested member or collection initializer.

When initializing an object with a `required` member, you must directly assign the member a value. You cannot use a nested member or collection initializer to set properties of the `required` member without first instantiating it.

Example

The following sample generates CS9036:

C#

```
class C
{
    public string? Prop { get; set; }
}

class Program
{
    public required C C { get; set; }

    static void Main()
    {
        var program = new Program()
        {
            // error CS9036: Required member 'Program.C' must be assigned a value,
it cannot use a nested member or collection initializer.
            C = { Prop = "a" }
        };
    }
}
```

Solution

To fix this error, directly assign a new instance of the required property and initialize its members:

C#

```
class C
{
    public string? Prop { get; set; }
```

```
}

class Program
{
    public required C C { get; set; }

    static void Main()
    {
        var program = new Program()
        {
            // Correct: Assign a new instance of C and then initialize its Prop
property
            C = new C { Prop = "a" }
        };
    }
}
```

For more information on required members, see the required modifier reference article and Object and Collection Initializers guide.

ⓘ **Note:** This article contains content created with AI. Learn more

Compiler Error CS9043

Article • 10/01/2022

Ref returning properties cannot be required.

The `required` modifier specifies that a member is required to be set during object initialization (i.e., via an object initializer.) For a property to be set within an object initializer, it must have a `set` accessor (a setter). `ref`-returning properties cannot have a setter and thus cannot also include the `required` modifier.

Example

The following sample generates CS9043:

C#

```
// CS9043.cs (5,29)

class C
{
    private int i;
    public required ref readonly int Number => ref i;
}
```

To correct this error

To have a `required` property, refactoring the property to return by value corrects this error:

C#

```
    public required int Number
    {
        get
        {
            return i;
        }
        set
        {
            i = value;
        }
    }
```

Compiler Warning (level 1) CS0183

Article • 09/15/2021

The given expression is always of the provided ('type') type

If a conditional statement always evaluates to **true**, then you do not need a conditional statement. This warning occurs when you try to evaluate a type using the **is** operator. If the evaluation is a value type, then the check is unnecessary.

The following sample generates CS0183:

C#

```
// CS0183.cs
// compile with: /W:1
using System;
public class Test
{
   public static void F(Int32 i32, String str)
   {
      if (str is Object)          // OK
         Console.WriteLine( "str is an object" );
      else
         Console.WriteLine( "str is not an object" );

      if (i32 is Object)   // CS0183
         Console.WriteLine( "i32 is an object" );
      else
         Console.WriteLine( "i32 is not an object" ); // never reached
   }

   public static void Main()
   {

      F(0, "CS0183");
      F(120, null);
   }
}
```

Compiler Warning (level 1) CS0184

Article • 09/15/2021

The given expression is never of the provided ('type') type

The expression can never be **true** because the variable you are testing is neither declared as ***type*** nor derived from ***type***.

The following sample generates CS0184:

```
C#
```

```
// CS0184.cs
// compile with: /W:1
class MyClass
{
   public static void Main()
   {
      int i = 0;
      if (i is string)   // CS0184
         i++;
   }
}
```

Compiler Warning (level 1) CS0197

Article • 03/11/2022

Passing 'argument' as ref or out or taking its address may cause a runtime exception because it is a field of a marshal-by-reference class

Any class that derives, directly or indirectly, from MarshalByRefObject is a marshal-by-reference class. Such a class can be marshalled by reference across process and machine boundaries. Thus, instances of this class could be proxies to remote objects. You cannot pass a field of a proxy object as ref or out. So, you cannot pass fields of such a class as `ref` or `out`, unless the instance is this, which can not be a proxy object.

Example

The following sample generates CS0197.

C#

```
// CS0197.cs
// compile with: /W:1
class X : System.MarshalByRefObject
{
   public int i;
}

class M
{
   public int i;
   static void AddSeventeen(ref int i)
   {
      i += 17;
   }

   static void Main()
   {
      X x = new X();
      x.i = 12;
      AddSeventeen(ref x.i);   // CS0197

      // OK
      M m = new M();
      m.i = 12;
      AddSeventeen(ref m.i);
   }
}
```

Compiler Warning (level 1) CS0420

Article • 09/15/2021

'identifier': a reference to a volatile field will not be treated as volatile

A volatile field should not normally be passed using a `ref` or **out** parameter, since it will not be treated as volatile within the scope of the function. There are exceptions to this, such as when calling an interlocked API. As with any warning, you may use the #pragma warning to disable this warning in those rare cases where you are intentionally using a volatile field as a reference parameter.

The following sample generates CS0420:

C#

```
// CS0420.cs
// compile with: /W:1
using System;

class TestClass
{
   private volatile int i;

   public void TestVolatile(ref int ii)
   {
   }

   public static void Main()
   {
      TestClass x = new TestClass();
      x.TestVolatile(ref x.i);   // CS0420
   }
}
```

Compiler Warning (level 1) CS0465

Article • 09/15/2021

Introducing a 'Finalize' method can interfere with destructor invocation. Did you intend to declare a destructor?

This warning occurs when you create a class with a method whose signature is `public virtual void Finalize`.

If such a class is used as a base class and if the deriving class defines a finalizer, the finalizer will override the base class `Finalize` method, not Finalize.

Example

The following sample generates CS0465.

C#

```
// CS0465.cs
// compile with: /target:library
class A
{
   public virtual void Finalize() {}   // CS0465
}

// OK
class B
{
   ~B() {}
}
```

Compiler Warning (level 1) CS0602

Article • 09/15/2021

The feature 'old_feature' is deprecated. Please use 'new_feature' instead

A language feature used in your code (*old_feature*) is still supported, but that support may be removed in a future release. Instead, you should use the recommended syntax (*new_feature*).

> **Note**
>
> This compiler warning is no longer used in Roslyn.

Compiler warning (level 1) CS0612

Article • 09/15/2021

'member' is obsolete

The class designer marked a member with the Obsolete attribute. This means that the member might not be supported in a future version of the class.

The following sample shows how accessing an obsolete member generates CS0612:

C#

```
// CS0612.cs
// compile with: /W:1
using System;

class MyClass
{
   [Obsolete]
   public static void ObsoleteMethod()
   {
   }

   [Obsolete]
   public static int ObsoleteField;
}

class MainClass
{
   static public void Main()
   {
      MyClass.ObsoleteMethod();    // CS0612 here: method is deprecated
      MyClass.ObsoleteField = 0;   // CS0612 here: field is deprecated
   }
}
```

Compiler Warning (level 1) CS0626

Article • 09/15/2021

Method, operator, or accessor 'method' is marked external and has no attributes on it. Consider adding a DllImport attribute to specify the external implementation.

A method marked `extern` should also be marked with an attribute, for example, the DllImport attribute.

The attribute specifies where the method is implemented. At run time, the program will need this information.

The following sample generates CS0626:

C#

```
// CS0626.cs
// compile with: /warnaserror
using System.Runtime.InteropServices;

public class MyClass
{
   static extern public void M(); // CS0626
   // try the following line
   // [DllImport("mydll.dll")] static extern public void M();

   public static void Main()
   {
   }
}
```

Compiler Warning (level 1) CS0672

Article • 09/15/2021

Member 'member1' overrides obsolete member 'member2. Add the Obsolete attribute to 'member1'

The compiler found an `override` to a method marked as `obsolete`. However, the overriding method was not itself marked as obsolete. The overriding method will still generate CS0612, if called.

Review your method declarations and explicitly indicate whether a method (and all of its overrides) should be marked `obsolete`.

The following sample generates CS0672:

C#

```
// CS0672.cs
// compile with: /W:1
class MyClass
{
   [System.Obsolete]
   public virtual void ObsoleteMethod()
   {
   }
}

class MyClass2 : MyClass
{
   public override void ObsoleteMethod()   // CS0672
   {
   }
}

class MainClass
{
   static public void Main()
   {
   }
}
```

Compiler Warning (level 1) CS0684

Article • 09/15/2021

'interface' interface marked with 'CoClassAttribute' not marked with 'ComImportAttribute'

If you specify **CoClassAttribute** on an interface, you must also specify **ComImportAttribute**.

The following sample generates CS0684:

C#

```
// CS0684.cs
// compile with: /W:1
using System;
using System.Runtime.InteropServices;

[CoClass(typeof(C))] // CS0684
// try the following line instead
// [CoClass(typeof(C)), ComImport]
interface I
{
}

class C
{
   static void Main() {}
}
```

Compiler Warning (level 1) CS0688

Article • 09/15/2021

'method1' has a link demand, but overrides or implements 'method2' which does not have a link demand. A security hole may exist.

The link demand set up on the derived class method can easily be circumvented by calling the base class method. To close the security hole, the base class method needs to also use the link demand. For more information, see Demand vs. LinkDemand.

Example

The following sample generates CS0688. To resolve the warning without modifying the base class, remove the security attribute from the overriding method. This will not solve the security problem.

C#

```
// CS0688.cs
// compile with: /W:1
using System;
using System.Security.Permissions;

class Base
{
    //Uncomment the following line to close the security hole
    //[FileIOPermission(SecurityAction.LinkDemand, All=@"C:\\")]
    public virtual void DoScaryFileStuff()
    {
    }
}

class Derived: Base
{
    [FileIOPermission(SecurityAction.LinkDemand, All=@"C:\\")] // CS0688
    public override void DoScaryFileStuff()
    {
    }
    static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS0809

Article • 08/13/2024

Obsolete member 'memberA' overrides non-obsolete member 'memberB'.

Typically, a member that is marked as obsolete should not override a member that is not marked as obsolete. This warning is generated in Visual Studio 2008 but not in Visual Studio 2005.

To correct this error

Remove the `Obsolete` attribute from the overriding member, or add it to the base class member.

Example

C#

```
// CS0809.cs
public class Base
{
    public virtual void Test1()
    {
    }
}
public class C : Base
{
    [System.Obsolete()]
    public override void Test1() // CS0809
    {
    }
}
```

Methods recognized as obsolete

Note that the compiler warning `CS0809` will lead to no CS0618 warning when actually calling the obsolete method.
In the following example compiler will not warn about the method `Test` being obsolete, because the declaration that is recognized by compiler when calling is in the base class `Base`, not the derived class `Derived`:

C#

```
class Base
{
    public virtual void Test() {}
}

class Derived : Base
{
    [System.Obsolete()]
    public override void Test() { }
}

static class Program
{
    public static void Main()
    {
        Derived derived = new();
        b.Foo();    // No CS0618
    }
}
```

To fix this add the `Obsolete` attribute to base class as well.

See also

- CS0618
- CS0672

Compiler Warning (level 1) CS1058

Article • 09/15/2021

A previous catch clause already catches all exceptions. All exceptions thrown will be wrapped in a System.Runtime.CompilerServices.RuntimeWrappedException

This attribute causes CS1058 if a `catch()` block has no specified exception type after a `catch (System.Exception e)` block. The warning advises that the `catch()` block will not catch any exceptions.

A `catch()` block after a `catch (System.Exception e)` block can catch non-CLS exceptions if the `RuntimeCompatibilityAttribute` is set to false in the AssemblyInfo.cs file: `[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = false)]`. If this attribute is not set explicitly to false, all thrown non-CLS exceptions are wrapped as Exceptions and the `catch (System.Exception e)` block catches them. For more information, see How to catch a non-CLS exception.

Example

The following example generates CS1058.

C#

```
// CS1058.cs
// CS1058 expected
using System.Runtime.CompilerServices;

// the following attribute is set to true by default in the C# compiler
// set to false in your source code to resolve CS1058
[assembly: RuntimeCompatibilityAttribute(WrapNonExceptionThrows = true)]

class TestClass
{
   static void Main()
   {
      try {}

      catch (System.Exception e) {
         System. Console.WriteLine("Caught exception {0}", e);
      }

      catch {}   // CS1058. This line will never be reached.
   }
}
```

Compiler Warning (level 1) CS1060

Article • 11/05/2021

Use of possibly unassigned field 'name'. Struct instance variables are initially unassigned if struct is unassigned.

Struct members are initialized to their default value if you do not explicitly initialize them. The default value for class types (and other reference types) is null. If the class is not initialized before any attempt to access it, a `NullReferenceException` will be thrown at run time. The compiler cannot determine definitively whether the class member will be initialized or not, and so CS1060 is a warning and not an error.

To correct this error

1. Provide a constructor for the `struct` that initializes all its members.

Example

The following code generates CS1060 because the class type `U` is a member of the `struct S` but is never initialized.

C#

```
// cs1060.cs
namespace CS1060
{
    public class U
    {
        public int i;
    }

    public struct S
    {
        public U u;
        // Add constructor to correct the error.
        //public S(int val)
        //{
        //    u = new U() { i = val };
        //}
    }
    public class Test
    {
        static void Main()
        {
            S s;
```

```
            s.u.i = 5;  // CS1060

            //Try these lines instead, and uncomment the constructor in S
            // S s = new S(0);
            // s.u.i = 5;

        }
    }
}
```

See also

- Structure types

Compiler Warning (level 1) CS1200

Article • 09/15/2021

The feature 'invalid feature' is deprecated. Please use 'valid feature' instead'.

The feature you are attempting to use is now deprecated. Update your code to use the valid feature instead.

Compiler Warning (level 1) CS1201

Article • 09/15/2021

The feature 'invalid feature' is deprecated. Please use 'valid feature' instead.

The feature you are attempting to use is now deprecated. Update your code to use the valid feature instead.

Compiler Warning (level 1) CS1202

Article • 09/15/2021

The feature 'invalid feature' is deprecated. Please use 'valid feature' instead'.

The feature you are attempting to use is now deprecated. Update your code to use the valid feature instead.

Compiler Warning (level 1) CS1203

Article • 09/15/2021

The feature 'feature' is deprecated. Please use 'feature' instead'.

The feature 'invalid feature' is deprecated. Please use 'valid feature' instead'.

The feature you are attempting to use is now deprecated. Update your code to use the valid feature instead.

Compiler Warning (level 1) CS1522

Article • 09/15/2021

Empty switch block

The compiler detected a switch block with no **case** or **default** statement. A `switch` block must have one or more **case** or **default** statements.

The following sample generates CS1522:

C#

```
// CS1522.cs
// compile with: /W:1
using System;
class x
{
   public static void Main()
   {
      int i = 6;

      switch(i)   // CS1522
      {
         // add something to the switch block, for example:
         /*
         case (5):
            Console.WriteLine("5");
            return;
         default:
            Console.WriteLine("not 5");
            return;
         */
      }
   }
}
```

Compiler Warning (level 1) CS1570

Article • 09/15/2021

XML comment on 'construct' has badly formed XML — 'reason'

When using **DocumentationFile**, any comments in the source code must be in XML. Any error with your XML markup will generate CS1570. For example:

- If you are passing a string to a **cref**, such as in an <exception> tag, the string must be enclosed in double quotation marks.
- If you're using a tag that doesn't have a closing tag, such as <seealso>, you must specify a forward slash before the closing angle bracket.
- If you need to use a greater-than or less-than symbol in the text of description, you need to represent them with `>` or `<`. Alternatively, you can use CDATA.
- The file or path attribute on an <include> tag was missing or improperly formed.

The following sample generates CS1570:

C#

```
public static class CompareFive
{
    // the following line generates CS1570
    /// <summary> returns true if < 5 </summary>
    // try one of the following instead
    // /// <summary> returns true if &lt; 5 </summary>
    // /// <summary><![CDATA[ returns true if < 5 ]]></summary>
    public static bool LessThanFive(int x) => x < 5;
}
```

Compiler Warning (level 1) CS1574

Article • 09/15/2021

XML comment on 'construct' has syntactically incorrect cref attribute 'name'

A string passed to a cref tag, for example, within an <exception> tag, referred to a member that is not available within the current build environment. The string that you pass to a cref tag must be the syntactically correct name of a member or field.

For more information, see Recommended Tags for Documentation Comments.

The following sample generates CS1574:

C#

```
// CS1574.cs
// compile with: /W:1 /doc:x.xml
using System;

/// <exception cref="System.Console.WriteLin">An exception class.
</exception>   // CS1574
// instead, uncomment and try the following line
// /// <exception cref="System.Console.WriteLine">An exception class.
</exception>
class EClass : Exception
{
}

class TestClass
{
   public static void Main()
   {
      try
      {
      }
      catch(EClass)
      {
      }
   }
}
```

Compiler Warning (level 1) CS1580

Article • 09/15/2021

Invalid type for parameter 'parameter number' in XML comment cref attribute

When attempting to reference an overload form of a method, the compiler detected a syntax error. Typically, this indicates that the parameter name, and not the type, was specified. A malformed line will appear in the generated XML file.

The following sample generates CS1580:

C#

```
// CS1580.cs
// compile with: /W:1 /doc:x.xml
using System;

/// <seealso cref="Test(i)"/>   // CS1580
// try the following line instead
// /// <seealso cref="Test(int)"/>
public class MyClass
{
   /// <summary>help text</summary>
   public static void Main()
   {
   }

   /// <summary>help text</summary>
   public void Test(int i)
   {
   }

   /// <summary>help text</summary>
   public void Test(char i)
   {
   }
}
```

Compiler Warning (level 1) CS1581

Article • 09/15/2021

Invalid return type in XML comment cref attribute

When attempting to reference a method, the compiler detected an error due to an invalid return type.

Example

The following sample generates CS1581:

C#

```
// CS1581.cs
// compile with: /W:1 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
    /// <summary>help text</summary>
    public static void Main()
    {
    }

    /// <summary>help text</summary>
    public static explicit operator int(MyClass f)
    {
        return 0;
    }
}

/// <seealso cref="MyClass.explicit operator intt(MyClass)"/>  // CS1581
// try the following line instead
// /// <seealso cref="MyClass.explicit operator int(MyClass)"/>
public class MyClass2
{
}
```

Compiler Warning (level 1) CS1584

Article • 09/15/2021

XML comment on 'member' has syntactically incorrect cref attribute 'invalid_syntax'

One of the parameters passed to a tag for documentation comments has invalid syntax. For more information, see Recommended Tags for Documentation Comments.

Example

The following sample generates CS1584.

C#

```
// CS1584.cs
// compile with: /W:1 /doc:CS1584.xml
/// <remarks>Test class</remarks>
public class Test
{
   /// <remarks>Called in <see cref="Test.Mai()n"/>.</remarks>   // CS1584
   // try the following line instead
   // /// <remarks>Called in <see cref="Test.Main()"/>.</remarks>
   public static void Test2() {}

   /// <remarks>Main method</remarks>
   public static void Main() {}
}
```

Compiler Warning (level 1) CS1589

Article • 09/15/2021

Unable to include XML fragment 'fragment' of file 'file' -- reason

The syntax (*fragment*) of a <include> tag, which referenced a file (`file`), was incorrect for the specified ***reason***.

A malformed line will be placed in the generated XML file.

The following sample generates CS1589:

C#

```
// CS1589.cs
// compile with: /W:1 /doc:CS1589_out.xml

/// <include file='CS1589.doc' path='MyDocs/MyMembers[@name="test"]/' />
// CS1589
// try the following line instead
// /// <include file='CS1589.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
   public static void Main()
   {
   }
}
```

Compiler Warning (level 1) CS1590

Article • 09/15/2021

Invalid XML include element -- Missing file attribute

A path or doc attribute, passed to the <include> tag, was missing or incomplete.

The following sample generates CS1590:

C#

```
// CS1590.cs
// compile with: /W:1 /doc:x.xml

/// <include path='MyDocs/MyMembers[@name="test"]/*' />   // CS1590
// try the following line instead
// /// <include file='x.doc' path='MyDocs/MyMembers[@name="test"]/*' />
class Test
{
   public static void Main()
   {
   }
}
```

Compiler Warning (level 1) CS1592

Article • 09/15/2021

Badly formed XML in included comments file -- 'reason'

A problem, reported as ***reason***, was found in the file specified by the <include> tag.

Compiler Warning (level 1) CS1598

Article • 09/15/2021

XML parser could not be loaded for the following reason: 'reason'. The XML documentation file 'file' will not be generated.

The **DocumentationFile** option was specified, but the compiler could not find and load msxml3.dll. Make sure that the file msxml3.dll is installed and registered.

Compiler Warning (level 1) CS1607

07/22/2025

Assembly generation -- reason

A warning was generated from the assembly-creation phase of the compilation.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

If you are building a 64-bit application on a 32-bit operating system, you must ensure that 64-bit versions of all referenced assemblies are installed on the target operating system.

All x86-specific common language runtime (CLR) assemblies have 64-bit counterparts (every CLR assembly will exist on all operating systems). Therefore, you can safely ignore CS1607 for CLR assemblies.

You can ignore this warning if you encounter it when you create an AssemblyInformationalVersionAttribute. The informational version is a string that attaches additional version information to an assembly; this information is not used at run time. Although you can specify any text, a warning message appears on compilation if the string is not in the format that is used by the assembly version number, or if it is in that format but contains wildcard characters. This warning is harmless.

For more information, see Al.exe Tool Errors and Warnings.

Compiler Warning (level 1) CS1616

Article • 09/15/2021

Option 'option' overrides attribute 'attribute' given in a source file or added module

This warning occurs if the assembly attributes AssemblyKeyFileAttribute or AssemblyKeyNameAttribute found in source conflict with the **KeyFile** or **KeyContainer** command line option or key file name or key container specified in the Project Properties.

For the example below, assume you have a key file named `cs1616.snk`. Generate this file with the command line:

Console

```
sn -k CS1616.snk
```

The following sample generates CS1616:

C#

```
// CS1616.cs
// compile with: /keyfile:cs1616.snk
using System.Reflection;

// To fix the error, remove the next line
[assembly: AssemblyKeyFile("cs1616b.snk")]  // CS1616

class C
{
  public static void Main()
  {
  }
}
```

Compiler Warning (level 1) CS1645

Article • 09/15/2021

Feature 'feature' is not part of the standardized ISO C# language specification, and may not be accepted by other compilers

The feature you are using is not part of the ISO standard. Code using this feature may not compile on other compilers.

C#

```
// CS1645.cs
// compile with: /W:1 /t:module /langversion:ISO-1
[assembly:System.CLSCompliant(false)]
// To suppress the warning use the switch: /nowarn:1645
[module:System.CLSCompliant(false)]   // CS1645
class Test
{
}
```

Compiler Warning (level 1) CS1658

Article • 09/15/2021

'warning text'. See also error 'error code'

The compiler emits this warning when it overrides an error with a warning. For information about the problem, refer to the error mentioned. To find the appropriate error from within the Visual Studio IDE, use the index. For example, if the text above reads "See also error 'CS1037'," look for CS1037 in the index.

Example

The following example generates CS1658.

C#

```
// CS1658.cs
// compile with: /doc:x.xml
// CS1584 expected
/// <summary>
/// </summary>
public class C
{
    /// <see cref="C.F(params object[])"/>  // CS1658
    public static void M()
    {
    }

    /// <summary>
    /// </summary>
    public void F(params object[] o)
    {
    }

    static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS1682

Article • 09/15/2021

Reference to type 'type' claims it is nested within 'nested type', but it could not be found

This error arises when you import references that do not agree with other references or with code you have written. A common way to get this error is to write code that refers to a class in metadata, and then you either delete that class or modify its definition.

Example 1

C#

```
// CS1682.cs
// compile with: /target:library /keyfile:mykey.snk
public class A {
   public class N1 {}
}
```

Example 2

C#

```
// CS1682_b.cs
// compile with: /target:library /reference:CS1682.dll
using System;
public class Ref {

   public static A A1() {
      return new A();
   }

   public static A.N1 N1() {
      return new A.N1();
   }
}
```

Example 3

C#

```
// CS1682_c.cs
// compile with: /target:library /keyfile:mykey.snk /out:CS1682.dll
```

```
public class A {
   public void M1() {}
}
```

Example 4

The following sample generates CS1682.

C#

```
// CS1682_d.cs
// compile with: /reference:CS1682.dll /reference:CS1682_b.dll /W:1
// CS1682 expected
class Tester {
   static void Main()
   {
      Ref.A1().M1();
   }
}
```

Compiler Warning (level 1) CS1684

Article • 09/15/2021

Reference to type 'Type Name' claims it is defined in 'Namespace', but it could not be found

This error can be caused by a reference inside one namespace referring to a type that it says exists inside a second namespace, but the type does not exist. For example, mydll.dll says that type `A` exists inside yourdll.dll, but no such type exists inside yourdll.dll. One possible cause of this error is that the version of yourdll.dll you are using is too old and `A` has not yet been defined.

The following sample generates CS1684.

Example 1

C#

```
// CS1684_a.cs
// compile with: /target:library /keyfile:CS1684.key
public class A {
   public void Test() {}
}

public class C2 {}
```

Example 2

C#

```
// CS1684_b.cs
// compile with: /target:library /r:cs1684_a.dll
// post-build command: del /f CS1684_a.dll
using System;
public class Ref
{
   public static A GetA() { return new A(); }
   public static C2 GetC() { return new C2(); }
}
```

Example 3

We now rebuild the first assembly, leaving out the definition of the class C2 not to be defined in the recompilation.

C#

```
// CS1684_c.cs
// compile with: /target:library /keyfile:CS1684.key /out:CS1684_a.dll
public class A {
   public void Test() {}
}
```

Example 4

This module references the second module by means of the identifier `Ref`. But the second module contains a reference to the class `C2`, which no longer exists because of the compilation in the previous step, and therefore the CS1684 error message is returned from the compilation of this module.

C#

```
// CS1684_d.cs
// compile with: /reference:cs1684_a.dll /reference:cs1684_b.dll
// CS1684 expected
class Tester
{
   public static void Main()
   {
      Ref.GetA().Test();
   }
}
```

Compiler Warning (level 1) CS1685

Article • 09/15/2021

The predefined type 'System.type name' is defined in multiple assemblies in the global alias; using definition from 'File Name'

This error occurs when a predefined system type such as System.int32 is found in two assemblies. One way this can happen is if you are referencing mscorlib from two different places, such as trying to run the .NET Framework versions 1.0 and 1.1 side-by-side.

The compiler will use the definition from only one of the assemblies. The compiler searches only global aliases, does not search libraries defined **/reference**. If you have specified **/nostdlib**, the compiler will search for Object, and in the future start all searches for predefined types in the file where it found Object.

Compiler Warning (level 1) CS1687

Article • 09/15/2021

Source file has exceeded the limit of 16,707,565 lines representable in the PDB, debug information will be incorrect

The PDB and debugger have some limitations about how big a file can be. If the source file is too big, the debugger will not behave properly beyond that limit. The user should either not emit debug information for that file by possibly using `#line hidden`, or they should find a way to shrink the file, possibly by splitting the file into multiple files. They might want to use the `partial` keyword to split up a large class.

Compiler Warning (level 1) CS1690

Article • 03/11/2022

Accessing a member on 'member' may cause a runtime exception because it is a field of a marshal-by-reference class

This warning occurs when you try to call a method, property, or indexer on a member of a class that derives from MarshalByRefObject, and the member is a value type. Objects that inherit from `MarshalByRefObject` are typically intended to be marshalled by reference across an application domain. If any code ever attempts to directly access the value-type member of such an object across an application domain, a runtime InvalidOperationException will occur. To resolve the warning, first copy the member into a local variable and call the method on that variable.

The following sample generates CS1690:

C#

```
// CS1690.cs
using System;

class WarningCS1690 : MarshalByRefObject
{
    int i = 5;

    public static void Main()
    {
        AppDomain domain = AppDomain.CreateDomain("MyDomain");
        Type t = typeof(WarningCS1690);
        WarningCS1690 e =
(WarningCS1690)domain.CreateInstanceAndUnwrap(t.Assembly.FullName,t.FullName
);

        e.i.ToString(); // CS1690

        // OK
        int i = e.i;
        i.ToString();
        e.i = i;
    }
}
```

Compiler Warning (level 1) CS1697

Article • 09/15/2021

Different checksum values given for 'file name'

You have specified more than one checksum for a given file. The debugger uses the checksum value to determine which file to debug when there is more than one file in a project with the same name. Most users will not encounter this error, but if you are writing an application that generates code, you may run into it. To resolve this error, ensure that you generate the checksum only once for any given code file.

Compiler Warning (level 1) CS1699

Article • 09/15/2021

Use command line option "compiler_option" or appropriate project settings instead of "attribute_name"

In order to sign an assembly, it is necessary to specify a key file. Prior to C# 2.0, you specified the key file using CLR attributes in source code. These attributes are now deprecated.

Beginning with C# 2.0, you should use the **Signing Page** of the **Project Designer** or the Assembly Linker to specify the key file.

The **Signing Page** of the **Project Designer** is the preferred method; for more information, see Signing Page, Project Designer and Managing Assembly and Manifest Signing.

The How to: Sign an Assembly with a Strong Name uses the following compiler options:

- **KeyFile** (C# Compiler Options) instead of the AssemblyKeyFileAttribute attribute.
- **KeyContainer** (C# Compiler Options) instead of AssemblyKeyNameAttribute.
- **DelaySign** (C# Compiler Options) instead of AssemblyDelaySignAttribute.

These attributes have been deprecated for the following reasons:

- There were security issues due to the attributes being embedded in the binary files produced by the compiler. Everyone who had your binary also had the keys stored in it.
- There were usability issues due to the fact that the path specified in the attributes was relative to the current working directory, which could change in the integrated development environment (IDE), or to the output directory. Thus, most times the key file is likely to be ..\..\mykey.snk. Attributes also make it more difficult for the project system to properly sign satellite assemblies. When you use the compiler options instead of these attributes, you can use a fully qualified path and file name for the key without anything being embedded in the output file; the project system and source code control system can properly manipulate that full path when projects are moved around; the project system can maintain a project-relative path to the key file, and still pass a full path to the compiler; other build programs can more easily sign outputs by passing the proper path directly to the compiler instead of generating a source file with the correct attributes.

- Using attributes with friend assemblies can hamper compiler efficiency. When you use attributes, the compiler does not know what the key is when it has to decide whether or not to grant friendship and so it has to guess. At the end of compilation, the compiler is able to verify the guess once it finally knows the key. When the key file is specified with a compiler option, the compiler can immediately decide whether to grant friendship.

Example

The following sample generates CS1699. To resolve the error, remove the attribute and compile with **/delaysign**.

C#

```
// CS1699.cs
// compile with: /target:library
[assembly:System.Reflection.AssemblyDelaySign(true)]   // CS1699
```

See also

- Signing Page, Project Designer
- Managing Assembly and Manifest Signing
- How to: Sign an Assembly with a Strong Name

Compiler Warning (level 1) CS1707

Article • 09/15/2021

Delegate 'DelegateName' bound to 'MethodName1' instead of 'MethodName2' because of new language rules

C# 2.0 implements new rules for binding a delegate to a method. Additional information is considered that was not looked at in the past. This warning indicates that the delegate is now bound to a different overload of the method than it was previously bound to. You may wish to verify that the delegate really should be bound to 'MethodName1' instead of 'MethodName2'.

For a description of how the compiler determines which method to bind a delegate to, see Using Variance in Delegates.

Compiler Warning (level 1) CS1720

Article • 09/15/2021

Expression will always cause a System.NullReferenceException because the default value of 'generic type' is null

If you write an expression involving the default of a generic type variable that is a reference type (for example, a class), this error will occur. Consider the following expression:

```
C#

default(T).ToString()
```

Since `T` is a reference type, its default value is null, and so attempting to apply the ToString method to it will throw a NullReferenceException.

Example

The class reference constraint on type `T` ensures that `T` is a reference type.

The following sample generates CS1720.

```
C#

// CS1720.cs
using System;
public class Tester
{
    public static void GenericClass<T>(T t1) where T : class
    {
        Console.WriteLine(default(T).ToString());  // CS1720
    }
    public static void Main() {}
}
```

Compiler Warning (level 1) CS1723

Article • 03/22/2023

XML comment has cref attribute 'attribute' that refers to a type parameter

This error is generated for an XML comment in case of using a <see/> tag with cross reference (cref) to a type parameter instead of the existing type (whether user-defined or built-in) in the code. It's impossible to link to 'attribute' of generic types because at the moment of creating the documentation the future type given as 'attribute' is not yet known.

To solve this issue <typeparamref/> tag should be used.

Example

The following example contains a comment generating CS1723 as well as a reference that can be linked correctly.

C#

```
public class Point
{
}

// compile with: /t:library /doc:filename.XML
///<summary>A generic list class.</summary>
///uses <see cref="T" />      // CS1723
///and <see cref="Point" />   // No warning
public class List<T, Point>
{
}
```

This example shows how to correctly link both generic type `T` as well as already known user-defined `Point`

C#

```
public class Point
{
}

// compile with: /t:library /doc:filename.XML
///<summary>A generic list class.</summary>
///uses <typeparamref name="T" />  // No warning
///and <see cref="Point" />        // No warning
public class List<T, Point>
```

```
{
}
```

Compiler Warning (level 1) CS1762

Article • 09/15/2021

A reference was created to embedded interop assembly '<assembly1>' because of an indirect reference to that assembly from assembly '<assembly2>'. Consider changing the 'Embed Interop Types' property on either assembly.

You have added a reference to an assembly (assembly1) that has the `Embed Interop Types` property set to `True`. This instructs the compiler to embed interop type information from that assembly. However, the compiler cannot embed interop type information from that assembly because another assembly that you have referenced (assembly2) also references that assembly (assembly1) and has the `Embed Interop Types` property set to `False`.

> **Note**
>
> Setting the `Embed Interop Types` property on an assembly reference to `True` is equivalent to referencing the assembly by using the `/link` option for the command-line compiler.

To address this warning

- To embed interop type information for both assemblies, set the `Embed Interop Types` property on all references to assembly1 to `True`. For more information about how to set that property, see Walkthrough: Embedding Types from Managed Assemblies.
- To remove the warning, you can set the `Embed Interop Types` property of assembly1 to `False`. In this case, a primary interop assembly (PIA) provides interop type information.

See also

- **EmbedInteropAssembly** (C# Compiler Options)
- Interoperating with Unmanaged Code

Compiler Warning (level 1) CS1911

Article • 09/15/2021

Access to member 'name' through a 'base' keyword from an anonymous method, lambda expression, query expression, or iterator results in unverifiable code. Consider moving the access into a helper method on the containing type.

Calling virtual functions with the `base` keyword inside the method body of an iterator or anonymous methods will result in unverifiable code. Unverifiable code will fail to run in a partial trust environment.

One resolution for CS1911 is to move the virtual function call to a helper function.

Example

The following sample generates CS1911.

C#

```
// CS1911.cs
// compile with: /W:1
using System;

delegate void D();
delegate D RetD();

class B {
   protected virtual void M() {
      Console.WriteLine("B.M");
   }
}

class Der : B {
   protected override void M() {
      Console.WriteLine("D.M");
   }

   void Test() { base.M(); }
   D Test2() { return new D(base.M); }

   public D CallBaseM() {
      return delegate () { base.M(); };   // CS1911

      // try the following line instead
      // return delegate () { Test(); };
   }

   public RetD DelToBaseM() {
```

```
        return delegate () { return new D(base.M); };   // CS1911

        // try the following line instead
        // return delegate () { return Test2(); };
    }
}

class Program {
    public static void Main() {
        Der der = new Der();
        D d = der.CallBaseM();
        d();
        RetD rd = der.DelToBaseM();
        rd()();
    }
}
```

Compiler Warning (level 1) CS1956

Article • 11/05/2021

Member 'name' implements interface member 'name' in type 'type'. There are multiple matches for the interface member at run-time. It is implementation dependent which method will be called.

This warning can be generated when two interface methods are differentiated only by whether a particular parameter is marked with `ref` or with `out`. It is best to change your code to avoid this warning because it is not obvious or guaranteed which method is called at run time.

Although C# distinguishes between `out` and `ref`, the CLR sees them as the same. When deciding which method implements the interface, the CLR just picks one.

To avoid this warning

1. Give the compiler some way to differentiate the methods. For example, you can give them different names or provide an additional parameter on one of them.

Example

The following code generates CS1956 because the two `Test` methods in `Base` differ only by the `ref`/`out` modifier on the first parameter.

C#

```
// cs1956.cs
class Base<T, S>
{
    // This is the method that should be called.
    public virtual int Test(out T x) // CS1956
    {
        x = default(T);
        return 0;
    }

    // This is the "last" method and is the one that ends up being called
    public virtual int Test(ref S x)
    {
        return 1;
    }
}

interface IFace
```

```
{
    int Test(out int x);
}

class Derived : Base<int, int>, IFace
{
    static int Main()
    {
        IFace x = new Derived();
        int y;
        return x.Test(out y);
    }
}
```

Compiler Warning (Level 1) CS1957

Article • 09/15/2021

Member 'name' overrides 'method'. There are multiple override candidates at run-time. It is implementation dependent which method will be called.

Method parameters that vary only by whether they are `ref` or `out` cannot be differentiated at run-time.

To avoid this warning

1. Give one of the methods a different name or different number of parameters.

Example

The following code generates CS1957:

C#

```
// cs1957.cs
class Base<T, S>
{
    public virtual string Test(out T x) // CS1957
    {
        x = default(T);
        return "Base.Test";
    }
    public virtual void Test(ref S x) { }
}

class Derived : Base<int, int>
{
    public override string Test(out int x)
    {
        x = 0;
        return "Derived.Test";
    }

    static int Main()
    {
        int x;
        if (new Derived().Test(out x) == "Derived.Test")
            return 0;
        return 1;
    }
}
```

Compiler Warning (level 1) CS2002

Article • 09/15/2021

Source file 'file' specified multiple times

A source file name was passed to the compiler more than once. You can only specify a file once to the compiler to build an output file.

This warning cannot be suppressed by the -nowarn option.

The following sample generates CS2002:

C#

```
// CS2002.cs
// compile with: CS2002.cs
public class A
{
public static void Main(){}
}
```

To generate the error, compile the example with the command line:

Console

```
csc CS2002.cs CS2002.cs
```

Compiler Warning (level 1) CS2014

Article • 09/15/2021

Compiler option 'old option' is obsolete, please use 'new option' instead

The form of the compiler option is deprecated. See C# Compiler Options for more information.

Compiler Warning (level 1) CS2023

Article • 05/13/2022

Ignoring /noconfig option because it was specified in a response file

The -noconfig compiler option was specified in a response file, which is not allowed.

This warning cannot be suppressed by the **NoWarn** option.

Compiler Warning (level 1) CS2029

Article • 05/13/2022

Invalid value for '/define'; 'identifier' is not a valid identifier

This warning occurs if the value that is used in the **DefineConstants** option has some invalid characters.

This warning cannot be suppressed by the **NoWarn** option.

Compiler Warning (level 1) CS3000

Article • 09/15/2021

Methods with variable arguments are not CLS-compliant

The arguments used in the method expose features that are not in the Common Language Specifications (CLS). For more information on CLS Compliance, see Language independence and language-independent components.

The following example generates the warning CS3000.

C#

```
// CS3000.cs
// compile with: /target:library
// CS3000 expected
[assembly:System.CLSCompliant(true)]

public class Test
{
   public void AddABunchOfInts( __arglist ) {}   // CS3000
}
```

Compiler Warning (level 1) CS3001

Article • 09/15/2021

Argument type 'type' is not CLS-compliant

A public, protected, or `protected internal` method must accept a parameter whose type is compliant with the Common Language Specification (CLS). For more information on CLS compliance, see Language independence and language-independent components.

Example

The following example generates CS3001:

C#

```
// CS3001.cs

[assembly:System.CLSCompliant(true)]
public class a
{
    public void bad(ushort i)   // CS3001
    {
    }

    private void OK(ushort i)   // OK, private method
    {
    }

    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3002

Article • 09/15/2021

Return type of 'method' is not CLS-compliant

A public, protected, or `protected internal` method must return a value whose type is compliant with the Common Language Specification (CLS). For more information on CLS compliance, see Language independence and language-independent components.

Example

The following example generates CS3002:

C#

```
// CS3002.cs

[assembly:System.CLSCompliant(true)]
public class a
{
    public ushort bad()   // CS3002, public method
    {
        ushort a;
        a = ushort.MaxValue;
        return a;
    }

    private ushort OK()   // OK, private method
    {
        ushort a;
        a = ushort.MaxValue;
        return a;
    }

    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3003

Article • 09/15/2021

Type of 'variable' is not CLS-compliant

A public, protected, or `protected internal` variable must be of a type that is compliant with the Common Language Specification (CLS). For more information on CLS Compliance, see Language independence and language-independent components.

Example

The following example generates CS3003:

C#

```
// CS3003.cs

[assembly:System.CLSCompliant(true)]
public class a
{
    public ushort a1;   // CS3003, public variable
    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3004

Article • 09/15/2021

Mixed and decomposed Unicode characters are not CLS-compliant

Only composed UNICODE characters are allowed in public, protected, or `protected internal` identifiers in order to be compliant with the Common Language Specification (CLS). For more information on CLS Compliance, see Language independence and language-independent components.

Compiler Warning (level 1) CS3005

Article • 09/15/2021

Identifier 'identifier' differing only in case is not CLS-compliant

A public, protected, or `protected internal` identifier, which differs from another `public`, `protected`, or `protected internal` identifier only in the case of one or more letters, is not compliant with the Common Language Specification (CLS). For more information on CLS compliance, see Language independence and language-independent components.

Example

The following example generates CS3003:

C#

```
// CS3005.cs

using System;

[assembly:CLSCompliant(true)]
public class a
{
    public static int a1 = 0;
    public static int A1 = 1;   // CS3005

    public static void Main()
    {
        Console.WriteLine(a1);
        Console.WriteLine(A1);
    }
}
```

Compiler Warning (level 1) CS3008

Article • 09/15/2021

Identifier 'identifier' differing only in case is not CLS-compliant

A public, protected, or `protected internal` identifier breaks compliance with the Common Language Specification (CLS) if it begins with an underscore character (_). For more information on CLS compliance, see Language independence and language-independent components.

Example

The following example generates CS3008:

C#

```
// CS3008.cs

using System;

[assembly:CLSCompliant(true)]
public class a
{
    public static int _a = 0;  // CS3008
    // OK, private
    // private static int _a1 = 0;

    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3009

Article • 09/15/2021

'type': base type 'type' is not CLS-compliant

A base type was marked as not having to be compliant with the Common Language Specification (CLS) in an assembly that was marked as being CLS compliant. Either remove the attribute that specifies the assembly is CLS compliant or remove the attribute that indicates the type is not CLS compliant. For more information on CLS Compliance, see Language independence and language-independent components.

Example

The following example generates CS3009:

C#

```
// CS3009.cs

using System;

[assembly:CLSCompliant(true)]
[CLSCompliant(false)]
public class B
{
}

public class C : B   // CS3009
{
    public static void Main () {}
}
```

Compiler Warning (level 1) CS3010

Article • 09/15/2021

'member': CLS-compliant interfaces must have only CLS-compliant members

In an assembly marked with `[assembly:CLSCompliant(true)]`, an interface contains a member marked with `[CLSCompliant(false)]`. Remove one of the Common Language Specification (CLS) compliance attributes. For more information about CLS Compliance, see Language independence and language-independent components.

Example

The following example generates CS3010:

C#

```
// CS3010.cs

using System;

[assembly:CLSCompliant(true)]
public interface I
{
    [CLSCompliant(false)]
    int M();   // CS3010
}

public class C : I
{
    public int M()
    {
        return 1;
    }

    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3011

Article • 09/15/2021

'member': only CLS-compliant members can be abstract

A class member cannot be both abstract and non-compliant with the Common Language Specification (CLS). The CLS specifies that all class members shall be implemented. For more information about CLS compliance, see Language independence and language-independent components.

Example

The following example generates CS3011:

C#

```
// CS3011.cs

using System;

[assembly:CLSCompliant(true)]
public abstract class I
{
    [CLSCompliant(false)]
    public abstract int M();   // CS3011

    // OK
    [CLSCompliant(false)]
    public void M2()
    {
    }
}

public class C : I
{
    public override int M()
    {
        return 1;
    }

    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3012

08/08/2025

You cannot specify the CLSCompliant attribute on a module that differs from the CLSCompliant attribute on the assembly

In order for a module to be compliant with the Common Language Specification (CLS) through `[module:System.CLSCompliant(true)]`, it must be built with the **module** element of the **OutputType** compiler option. For more information on the CLS, see Language independence and language-independent components.

Example

The following example, when built without `/target:module`, generates CS3012:

C#

```
// CS3012.cs
// compile with: /W:1

[module:System.CLSCompliant(true)]   // CS3012
public class C
{
    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3013

08/08/2025

Added modules must be marked with the CLSCompliant attribute to match the assembly

A module that was compiled with the **module** element of the **OutputType** compiler option was added to a compilation with **AddModule**. However, the module's compliance with the Common Language Specification (CLS) does not agree with the CLS state of the current compilation.

CLS compliance is indicated with the module attribute. For example, `[module:CLSCompliant(true)]` indicates that the module is CLS compliant, and `[module:CLSCompliant(false)]` indicates that the module is not CLS compliant. The default is `[module:CLSCompliant(false)]`. For more information on the CLS, see Language independence and language-independent components.

Compiler Warning (level 1) CS3014

Article • 09/15/2021

'member' does not need a CLSCompliant attribute because the assembly does not have a CLSCompliant attribute

In a source code file where compliance with the Common Language Specification (CLS) was not specified, a construct in the file was marked as being CLS compliant. This is not allowed. To resolve this warning, add an assembly level CLS compliant attribute to the file (in the following example, uncomment the line that contains the assembly level attribute). For more information about CLS compliance, see Language independence and language-independent components.

Example

The following example generates CS3014:

C#

```
// CS3014.cs

using System;

// [assembly:CLSCompliant(true)]
public class I
{
    [CLSCompliant(true)]   // CS3014
    public void M()
    {
    }

    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3015

Article • 09/15/2021

'method signature' has no accessible constructors which use only CLS-compliant types

To be compliant with the Common Language Specification (CLS), the argument list of an attribute class cannot contain an array. For more information on CLS compliance, see Language independence and language-independent components.

Example

The following sample generates C3015.

C#

```
// CS3015.cs
// compile with: /target:library
using System;

[assembly:CLSCompliant(true)]
public class MyAttribute : Attribute
{
   public MyAttribute(int[] ai) {}   // CS3015
   // try the following line instead
   // public MyAttribute(int ai) {}
}
```

Compiler Warning (level 1) CS3017

Article • 09/15/2021

You cannot specify the CLSCompliant attribute on a module that differs from the CLSCompliant attribute on the assembly

This warning occurs if you have a assembly CLSCompliant attribute that conflicts with a module CLSCompliant attribute. An assembly that is CLS compliant cannot contain modules that are not CLS compliant. To resolve this warning, make sure the assembly and module CLSCompliant attributes are either both true or both false, or remove one of the attributes. For more information on CLS compliance, see Language independence and language-independent components.

Example

The following example generates CS3017:

C#

```
// CS3017.cs
// compile with: /target:module

using System;

[module: CLSCompliant(true)]
[assembly: CLSCompliant(false)]  // CS3017
// Try this line instead:
// [assembly: CLSCompliant(true)]
class C
{
    static void Main() {}
}
```

Compiler Warning (level 1) CS3018

Article • 09/15/2021

'type' cannot be marked as CLS-Compliant because it is a member of non CLS-compliant type 'type'

This warning occurs if a nested class with the CLSCompliant attribute set to `true` is declared as a member of a class declared with the CLSCompliant attribute set to `false`. This is not allowed, since a nested class cannot be CLS-compliant if it is a member of an outer class that is not CLS-compliant. To resolve this warning, remove the CLSCompliant attribute from the nested class, or change it from `true` to `false`. For more information on CLS compliance, see Language independence and language-independent components.

Example

The following sample generates CS3018.

C#

```
// CS3018.cs
// compile with: /target:library
using System;

[assembly: CLSCompliant(true)]
[CLSCompliant(false)]
public class Outer
{
   [CLSCompliant(true)]   // CS3018
   public class Nested {}

   // OK
   public class Nested2 {}

   [CLSCompliant(false)]
   public class Nested3 {}
}
```

Compiler Warning (level 1) CS3022

Article • 09/15/2021

CLSCompliant attribute has no meaning when applied to parameters. Try putting it on the method instead.

Method parameters are not checked for CLS Compliance, since the CLS Compliance rules apply to methods and type declarations.

Example

The following sample generates CS3022:

C#

```
// CS3022.cs
// compile with: /W:1

using System;

[assembly: CLSCompliant(true)]
[CLSCompliant(true)]
public class C
{
    public void F([CLSCompliant(true)] int i)
    {
    }

    public static void Main()
    {
    }
}
```

Compiler Warning (level 1) CS3023

Article • 09/15/2021

CLSCompliant attribute has no meaning when applied to return types. Try putting it on the method instead.

Function return types are not checked for CLS Compliance, since the CLS Compliance rules apply to methods and type declarations.

Example

The following example generates warning CS3023:

C#

```
// C3023.cs

[assembly:System.CLSCompliant(true)]
public class Test
{
    [return:System.CLSCompliant(true)]  // CS3023
    // Try this instead:
    // [method:System.CLSCompliant(true)]
    public static int Main()
    {
        return 0;
    }
}
```

Compiler Warning (level 1) CS3024

Article • 09/15/2021

Constraint type 'type' is not CLS-compliant.

The compiler issues this warning because the use of a non-CLS-compliant type as a generic type constraint could make it impossible for code written in some languages to consume your generic class.

To eliminate this warning

1. Use a CLS-compliant type for the type constraint.

Example

The following example generates CS3024 in several locations:

C#

```
// cs3024.cs
// Compile with: /target:library
 [assembly: System.CLSCompliant(true)]

[type: System.CLSCompliant(false)]
public class TestClass // CS3024
{
    public ushort us;
}
[type: System.CLSCompliant(false)]
public interface ITest // CS3024
{}
public interface I<T> where T : TestClass
{}
public class TestClass_2<T> where T : ITest
{}
public class TestClass_3<T> : I<T> where T : TestClass
{}
public class TestClass_4<T> : TestClass_2<T> where T : ITest
{}
public class Test
{
    public static int Main()
    {
        return 0;
    }
}
```

See also

- Constraints on Type Parameters

Compiler Warning (level 1) CS3026

Article • 09/15/2021

CLS-compliant field 'field' cannot be volatile

A volatile variable should not be CLS compliant.

Example

The following example generates CS3026.

C#

```
// CS3026.cs
[assembly:System.CLSCompliant(true)]
public class Test
{
    public volatile int v0 =0;   // CS3026
    // To resolve remove the CLS-Compliant attribute.
    public static void Main() { }
}
```

Compiler Warning (level 1) CS3027

Article • 09/15/2021

'type_1' is not CLS-compliant because base interface 'type_2' is not CLS-compliant

A non-CLS compliant type cannot be a base type for a type that is CLS compliant.

Example 1

The following sample contains an interface with a method that uses a non-CLS compliant type in its signature, making the type non-CLS compliant.

C#

```
// CS3027.cs
// compile with: /target:library
public interface IBase
{
   void IMethod(uint i);
}
```

Example 2

The following sample generates CS3027.

C#

```
// CS3027_b.cs
// compile with: /reference:CS3027.dll /target:library /W:1
[assembly:System.CLSCompliant(true)]
public interface IDerived : IBase {}
```

Compiler Warning (level 1) CS5000

Article • 09/15/2021

Unknown compiler option '/option'

An invalid compiler option was specified.

> **Note**
>
> This compiler error is no longer used in Roslyn.

Compiler Warning (level 2) CS0108

Article • 02/21/2024

'member1' hides inherited member 'member2'. Use the new keyword if hiding was intended.

A member was declared with the same name as a member in a base class. However, the new modifier was not used.

The following sample generates CS0108. You can resolve CS0108 in one of two ways:

- Rename the member in the derived class if member hiding was not intended.
- Use the `new` modifier to declare that the derived member hiding of the base member was intentional.

C#

```
// CS0108.cs
// compile with: /W:2
using System;

namespace MyNamespace;

public class BaseClass
{
    public int i = 1;
}

public class DerivedClass : BaseClass
{
    public static int i = 2;   // CS0108
    // Use the following line instead:
    // public static new int i = 2;

    public static void Main()
    {
        Console.WriteLine(i);
    }
}
```

See also

- new Modifier

Compiler Warning (level 2) CS0114

Article • 09/15/2021

'function1' hides inherited member 'function2'. To make the current method override that implementation, add the override keyword. Otherwise add the new keyword.

A declaration in a class conflicts with a declaration in a base class such that the base class member will be hidden.

For more information, see base.

The following sample generates CS0114:

C#

```
// CS0114.cs
// compile with: /W:2 /warnaserror
abstract public class clx
{
   public abstract void f();
}

public class cly : clx
{
   public void f() // CS0114, hides base class member
   // try the following line instead
   // override public void f()
   {
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 2) CS0162

Article • 09/15/2021

Unreachable code detected

The compiler detected code that will never be executed.

Example

The following example generates CS0162:

C#

```
// CS0162.cs
// compile with: /W:2
public class Program
{
    public static void Main()
    {
        goto lab1;
        {
            // The following statements cannot be reached:
            int i = 9;   // CS0162
            i++;
        }
    lab1:
        {
        }
    }
}
```

Another common example where this error is generated is as follows:

C#

```
public static class Class1
{
    public static string Method1()
    {
        string x = "a";
        switch (x)
        {
            case "a":
                return "a";
                break;          // CS0162
        }
        return "";
```

```
    }
}
```

The `break` statement cannot be reached because it occurs after the `return` statement. The `return` statement ends the enclosing `case` branch.

Compiler Warning (level 2) CS0164

Article • 09/15/2021

This label has not been referenced

A label was declared but never used.

The following sample generates CS0164:

C#

```
// CS0164.cs
// compile with: /W:2
public class Program
{
    public static void Main()
    {
        int i = 0;
    l1: // CS0164
        i++;
        // Uncomment the following lines to resolve this warning.
        // if (i < 10)
        //     goto l1;
    }
}
```

Compiler Warning (level 2) CS0252

Article • 09/15/2021

Possible unintended reference comparison; to get a value comparison, cast the left hand side to type 'type'

The compiler is doing a reference comparison. If you want to compare the value of strings, cast the left side of the expression to `type`.

The following sample generates CS0252:

C#

```
// CS0252.cs
// compile with: /W:2
using System;

class MyClass
{
   public static void Main()
   {
      string s = "11";
      object o = s + s;

      bool b = o == s;   // CS0252
      // try the following line instead
      // bool b = (string)o == s;
   }
}
```

Compiler Warning (level 2) CS0253

Article • 09/15/2021

Possible unintended reference comparison; to get a value comparison, cast the right hand side to type 'type'

The compiler is doing a reference comparison. If you want to compare the value of strings, cast the right side of the expression to `type`.

The following sample generates CS0253:

C#

```
// CS0253.cs
// compile with: /W:2
using System;
class MyClass
{
   public static void Main()
   {
      string s = "11";
      object o = s + s;

      bool c = s == o;   // CS0253
      // try the following line instead
      // bool c = s == (string)o;
   }
}
```

Compiler Warning (level 2) CS0278

Article • 09/15/2021

'type' does not implement the 'pattern name' pattern. 'method name' is ambiguous with 'method name'.

There are several statements in C# that rely on defined patterns, such as `foreach` and `using`. For example, the foreach statement relies on the collection class implementing the "enumerable" pattern.

CS0278 can occur if the compiler is unable to make the match due to ambiguities. For example, the "enumerable" pattern requires that there be a method called `MoveNext`, and your code might contain two methods called `MoveNext`. The compiler will attempt to find an interface to use, but it is recommended that you determine and resolve the cause of the ambiguity.

Example

The following sample generates CS0278.

C#

```
// CS0278.cs
using System.Collections.Generic;
public class myTest
{
   public static void TestForeach<W>(W w)
      where W: IEnumerable<int>, IEnumerable<string>
   {
      foreach (int i in w) {}   // CS0278
   }
}
```

Compiler Warning (level 2) CS0279

Article • 09/15/2021

'type name' does not implement the 'pattern name' pattern. 'method name' is either static or not public.

There are several statements in C# that rely on defined patterns, such as `foreach` and `using`. For example, `foreach` relies on the collection class implementing the enumerable pattern. This error occurs when the compiler is unable to make the match due to a method being declared `static` or not `public`. Methods in patterns are required to be instances of classes, and to be public.

Example

The following example generates CS0279:

C#

```
// CS0279.cs

using System;
using System.Collections;

public class myTest : IEnumerable
{
    IEnumerator IEnumerable.GetEnumerator()
    {
        yield return 0;
    }

    internal IEnumerator GetEnumerator()
    {
        yield return 0;
    }

    public static void Main()
    {
        foreach (int i in new myTest()) {}  // CS0279
    }
}
```

Compiler Warning (level 2) CS0280

Article • 09/15/2021

'type' does not implement the 'pattern name' pattern. 'method name' has the wrong signature.

Two statements in C#, **foreach** and **using**, rely on predefined patterns, "collection" and "resource" respectively. This warning occurs when the compiler cannot match one of these statements to its pattern due to a method's incorrect signature. For example, the "collection" pattern requires that there be a method called MoveNext which takes no parameters and returns a `boolean`. Your code might contain a MoveNext method that has a parameter or perhaps returns an object.

The "resource" pattern and `using` provide another example. The "resource" pattern requires the Dispose method; if you define a property with the same name, you will get this warning.

To resolve this warning, ensure that the method signatures in your type match the signatures of the corresponding methods in the pattern, and ensure that you have no properties with the same name as a method required by the pattern.

Example

The following sample generates CS0280.

C#

```
// CS0280.cs
using System;
using System.Collections;

public class ValidBase: IEnumerable
{
   IEnumerator IEnumerable.GetEnumerator()
   {
      yield return 0;
   }

   internal IEnumerator GetEnumerator()
   {
      yield return 0;
   }
}

class Derived : ValidBase
{
```

```
    // field, not method
    new public int GetEnumerator;
}

public class Test
{
    public static void Main()
    {
        foreach (int i in new Derived()) {}   // CS0280
    }
}
```

Compiler Warning (level 2) CS0435

Article • 09/15/2021

The namespace 'namespace' in 'assembly' conflicts with the imported type 'type' in 'assembly'. Using the namespace defined in 'assembly'.

This warning is issued when a namespace in a source file (file_2) conflicts with an imported type in file_1. The compiler uses the one in the source file.

The following example generates CS0435:

Compile this file first:

C#

```
// CS0435_1.cs
// compile with: /t:library
public class Util
{
   public class A { }
}
```

Then, compile this file:

C#

```
// CS0435_2.cs
// compile with: /r:CS0435_1.dll

using System;

namespace Util
{
   public class A { }
}

public class Test
{
   public static void Main()
   {
      Console.WriteLine(typeof(Util.A)); // CS0435
   }
}
```

Compiler Warning (level 2) CS0436

10/09/2025

The type 'type' in 'assembly' conflicts with the imported type 'type2' in 'assembly'. Using the type defined in 'assembly'.

This warning occurs when a type defined in your source code has the same fully qualified name (namespace and type name) as a type imported from a referenced assembly. When this name conflict occurs, the compiler uses the locally defined type from your source file and ignores the imported type.

What constitutes a conflict

A conflict occurs when two types have identical fully qualified names, meaning:

- They have the same namespace
- They have the same type name
- They're both accessible in the current compilation context

The conflict is determined solely by the type's fully qualified name, not by its implementation details. Two types with the same name but different implementations (such as different methods, properties, or field values) still conflict. The compiler can't use both types simultaneously because they have the same identity.

Example

The following example demonstrates CS0436. In this scenario, a type `A` is defined in an external library and also locally in the source file. Even though the two types have different implementations (they print different strings), they conflict because they share the same fully qualified name.

First, create a library that defines type `A`:

C#

```
// CS0436_a.cs
// compile with: /target:library
public class A {
   public void Test() {
      System.Console.WriteLine("CS0436_a");
   }
}
```

Then, compile the following code that defines another type `A` and references the library. The compiler issues CS0436 because both types have the fully qualified name `A` (in the global namespace):

C#

```
// CS0436_b.cs
// compile with: /reference:CS0436_a.dll
// CS0436 expected
public class A {
   public void Test() {
      System.Console.WriteLine("CS0436_b");
   }
}

public class Test
{
   public static void Main()
   {
      A x = new A();
      x.Test();
   }
}
```

When you compile and run this code, the compiler uses the locally defined `A` (from CS0436_b.cs) and issues a warning. The output is:

Console

```
CS0436_b
```

Note that the conflict exists even though the two `A` types have different implementations. The difference in the string literal (`"CS0436_a"` versus `"CS0436_b"`) doesn't prevent the conflict. What matters is that both types have the same fully qualified name `A`.

How to resolve this warning

To resolve this warning, you can:

1. Rename one of the conflicting types.
2. Use a different namespace for one of the types.
3. Remove the reference to the assembly containing the conflicting type if it's not needed.
4. Use an extern alias to disambiguate between the two types if you need to use both (see CS0433 for examples of using extern aliases).

Compiler Warning (level 2) CS0437

Article • 09/15/2021

The type 'type' in 'assembly2' conflicts with the imported namespace 'namespace' in 'fassembly1'. Using the type defined in 'assembly'.

This warning is issued when a type in a source file, file_2, conflicts with an imported namespace in file_1. The compiler uses the type in the source file.

Example 1

C#

```
// CS0437_a.cs
// compile with: /target:library
namespace Util
{
   public class A {
      public void Test() {
         System.Console.WriteLine("CS0437_a.cs");
      }
   }
}
```

Example 2

The following sample generates CS0437.

C#

```
// CS0437_b.cs
// compile with: /reference:CS0437_a.dll /W:2
// CS0437 expected
class Util
{
   public class A {
      public void Test() {
         System.Console.WriteLine("CS0437_b.cs");
      }
   }
}

public class Test
{
   public static void Main()
   {
```

```
        Util.A x = new Util.A();
        x.Test();
    }
}
```

Compiler Warning (level 2) CS0444

Article • 09/15/2021

Predefined type 'type name 1' was not found in 'System namespace 1' but was found in 'System namespace 2'

A predefined object such as Int32 was not found where the compiler expected to find it, but instead found it in 'System namespace 2'.

The error could indicate that .NET is installed incorrectly. To fix this, reinstall .NET.

If you are writing your own base class libraries, you might also encounter this error. In this case, to resolve the error, rebuild mscorlib.

> **Note**
>
> This compiler warning is no longer used in Roslyn.

Compiler Warning (level 2) CS0458

Article • 09/15/2021

The result of the expression is always 'null' of type 'type name'

This warning is caused by a nullable value type expression that always results in `null`.

The following code generates warning CS0458.

Example

This example illustrates a number of the different operations with nullable value types that will cause this error.

C#

```
// CS0458.cs
using System;
public  class Test
{
    public static void Main()
    {
        int a = 5;
        int? b = a + null;     // CS0458
        int? qa = 15;
        b = qa + null;         // CS0458
        b -= null;             // CS0458
        int? qa2 = null;
        b = qa2 + null;        // CS0458
        qa2 -= null;           // CS0458
    }
}
```

Compiler Warning (level 2) CS0464

Article • 09/15/2021

Comparing with null of type 'type' always produces 'false'

This warning is produced when you perform a comparison between a nullable value type variable and null, and the comparison is not `==` or `!=`. To resolve this error, verify if you really want to check a value for `null`. A comparison like `i == null` can be either true of false. A comparison like `i > null` is always false.

Example

The following sample generates CS0464.

C#

```
// CS0464.cs
class MyClass
{
   public static void Main()
   {
      int? i = 0;
      if (i < null) ;   // CS0464

      i++;
   }
}
```

Compiler Warning (level 2) CS0467

Article • 09/15/2021

Ambiguity between method 'method' and non-method 'non-method'. Using method group.

Inherited members from different interfaces that have the same signature cause an ambiguity error.

Example

The following example generates CS0467.

C#

```
// CS0467.cs
interface IList
{
    int Count { get; set; }
}

interface ICounter
{
    void Count(int i);
}

interface IListCounter : IList, ICounter {}

class Driver
{
    void Test(IListCounter x)
    {
        // The following line causes the warning. The assignment also
        // causes an error because you can't assign a value to a method.
        x.Count = 1;
        x.Count(3);
        // To resolve the warning, you can change the name of the method or
        // the property.

        // You can also disambiguate by specifying IList or ICounter.
        ((IList)x).Count = 1;
        ((ICounter)x).Count(3);
    }

    static void Main()
    {
    }
}
```

Compiler Warning (level 2) CS0469

Article • 09/15/2021

The 'goto case' value is not implicitly convertible to type 'type'

When you use `goto case`, there must be an implicit conversion from the value of the goto case to the type of the switch.

Example

The following sample generates CS0469.

C#

```
// CS0469.cs
// compile with: /W:2
class Test
{
   static void Main()
   {
      char c = (char)180;
      switch (c)
      {
         case (char)127:
            break;

         case (char)180:
            goto case 127;   // CS0469
            // try the following line instead
            // goto case (char) 127;
      }
   }
}
```

Compiler Warning (level 2) CS0472

Article • 09/15/2021

The result of the expression is always 'value1' since a value of type 'value2' is never equal to 'null' of type 'value3'

The compiler should warn if you use an operator with a constant null value.

Example

The following sample generates CS0472.

C#

```
public class Test
{
    public static int Main()
    {
        int i = 5;
        int counter = 0;

        // Comparison:
        if (i == null)  // CS0472
        // To resolve, use a valid value for i.
            counter++;
        return counter;
    }
}
```

Compiler Warning (level 2) CS0618

Article • 05/04/2023

'member' is obsolete: 'text'

A class member was marked with the `Obsolete` attribute, such that a warning will be issued when the class member is referenced. For more information, see Assembly level attributes.

The following sample generates CS0618:

C#

```
// CS0618.cs
// compile with: /W:2
using System;

public class C
{
   [Obsolete("Use newMethod instead", false)]   // warn if referenced
   public static void m2()
   {
   }

   public static void newMethod()
   {
   }
}

class MyClass
{
   public static void Main()
   {
      C.m2();  // CS0618
   }
}
```

Compiler Warning (level 2) CS0652

Article • 09/15/2021

Comparison to integral constant is useless; the constant is outside the range of type 'type'

The compiler detected a comparison between a constant and a variable where the constant is out of the range of the variable.

The following sample generates CS0652:

C#

```
// CS0652.cs
// compile with: /W:2
public class Class1
{
   private static byte i = 0;
   public static void Main()
   {
      short j = 256;
      if (i == 256)   // CS0652, 256 is out of range for byte
         i = 0;
   }
}
```

Compiler Warning (level 2) CS1571

Article • 09/15/2021

XML comment on 'construct' has a duplicate param tag for 'parameter'

When using the **DocumentationFile** compiler option, multiple comments were found for the same method parameter. Remove one of the duplicate lines.

The following sample generates CS1571:

C#

```
// CS1571.cs
// compile with: /W:2 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   /// <param name='Char1'>An initial.</param>
   /// <param name='Int1'>Used to indicate status.</param> // CS1571
   public static void MyMethod(int Int1, char Char1)
   {
   }

   /// <summary>help text</summary>
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 2) CS1572

Article • 09/15/2021

XML comment on 'construct' has a param tag for 'parameter', but there is no parameter by that name

When using the **DocumentationFile** compiler option, a comment was specified for a parameter that does not exist for the method. Change the value passed to the name attribute or remove one of the comment lines.

The following sample generates CS1572:

C#

```
// CS1572.cs
// compile with: /W:2 /doc:x.xml

/// <summary>help text</summary>
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   /// <param name='Char1'>Used to indicate status.</param>
   /// <param name='Char2'>???</param> // CS1572
   public static void MyMethod(int Int1, char Char1)
   {
   }

   /// <summary>help text</summary>
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 2) CS1587

Article • 09/15/2021

XML comment is not placed on a valid language element

Recommended tags for documentation comments are not allowed on all language elements. For example, a tag is not allowed on a namespace. For more information on XML comments, see Recommended Tags for Documentation Comments.

Example

The following sample generates CS1587:

C#

```
// CS1587.cs
// compile with: /W:2 /doc:x.xml

/// <summary>test</summary>   // CS1587, tag not allowed on namespace
namespace MySpace
{
    class MyClass
    {
        public static void Main()
        {
        }
    }
}
```

Compiler Warning (level 2) CS1668

Article • 09/15/2021

Invalid search path 'path' specified in 'path string' -- 'system error message'

The path supplied to **AdditionalLibPaths** at the command line was not valid, or a path in the LIB environment variable is invalid. Check the path used to verify that it exists and can be accessed. The error message in single quotation marks is the error returned from the operating system.

See also

- C# Compiler Options
- C# Compiler Errors

Compiler Warning (level 2) CS1698

Article • 09/15/2021

Circular assembly reference 'AssemblyName1' does not match the output assembly name 'AssemblyName2'. Try adding a reference to 'AssemblyName1' or changing the output assembly name to match.

CS1698 occurs when an assembly reference is incorrect. This can happen if a referenced assembly is recompiled. To resolve, do not replace an assembly that itself is a dependency of an assembly you are referencing.

Example 1

C#

```
// CS1698_a.cs
// compile with: /target:library /keyfile:mykey.snk
[assembly:System.Reflection.AssemblyVersion("2")]
public class CS1698_a {}
```

Example 2

C#

```
// CS1698_b.cs
// compile with: /target:library /reference:CS1698_a.dll /keyfile:mykey.snk
public class CS1698_b : CS1698_a {}
```

Example 3

The following sample generates CS1698.

C#

```
// CS1698_c.cs
// compile with: /target:library /out:cs1698_a.dll /reference:cs1698_b.dll
/keyfile:mykey.snk
// CS1698 expected
[assembly:System.Reflection.AssemblyVersion("3")]
public class CS1698_c : CS1698_b {}
public class CS1698_a {}
```

Compiler Warning (level 2) CS1701

Article • 09/15/2021

Assuming assembly reference "Assembly Name #1" matches "Assembly Name #2", you may need to supply runtime policy

The two assemblies differ in release and/or version number. For unification to occur, you must specify directives in the application's .config file, and you must provide the correct strong name of an assembly, as demonstrated in the following example code.

Example 1

The following multifile sample references an assembly using two different external aliases. This first sample builds the older version of the code that creates assembly CS1701_d.

C#

```
// CS1701_a.cs
// compile with: /target:library /out:cs1701_d.dll /keyfile:mykey.snk
using System.Reflection;
[assembly:AssemblyVersion("1.0")]
public class A {
   public void M1() {}
}

public class C1 {}
```

Example 2

This is the code that creates the newer version of assembly CS1701_d. Note that it compiles into a different directory than the older version, necessary since the output files have the same names.

C#

```
// CS1701_b.cs
// compile with: /target:library /out:c:\\cs1701_d.dll /keyfile:mykey.snk
using System.Reflection;
[assembly:AssemblyVersion("2.0")]
public class A {
   public void M2() {}
   public void M1() {}
}
```

```
public class C2 {}
public class C1 {}
```

Example 3

This sample sets up the external aliases A1 and A2.

C#

```
// CS1701_c.cs
// compile with: /target:library /reference:A2=c:\\cs1701_d.dll
/reference:A1=cs1701_d.dll

extern alias A1;
extern alias A2;
// using System;
using a1 = A1::A;
using a2 = A2::A;

public class Ref {
   public static a1 A1() { return new a1(); }
   public static a2 A2() { return new a2(); }

   public static A1::C1 M1() { return new A1::C1(); }
   public static A2::C2 M2() { return new A2::C2(); }
}
```

Example 4

This sample calls methods using two different aliases of A. The following sample generates C1701.

C#

```
// CS1701_d.cs
// compile with: /reference:c:\\CS1701_d.dll /reference:CS1701_c.dll
// CS1701 expected
class Tester {
   public static void Main() {
      Ref.A1().M1();
      Ref.A2().M2();
   }
}
```

Compiler Warning (level 2) CS1710

Article • 09/15/2021

XML comment on 'type' has a duplicate typeparam tag for 'parameter'

The documentation of a generic type includes a duplicate tag for the type parameter.

Example

The following code will cause warning CS1710 to appear.

C#

```
// CS1710.cs
// compile with: /doc:cs1710.xml
// To resolve this warning, delete one of the duplicate <typeparam>'s.
using System;
class Stack<ItemType>
{
}

/// <typeparam name="MyType">can be an int</typeparam>
/// <typeparam name="MyType">can be an int</typeparam>
class MyStackWrapper<MyType>
{
    // Open constructed type Stack<MyType>.
    Stack<MyType> stack;
    public MyStackWrapper(Stack<MyType> s)
    {
        stack = s;
    }
}

class CMain
{
    public static void Main()
    {
        // Closed constructed type Stack<int>.
        Stack<int> stackInt = new Stack<int>();
        MyStackWrapper<int> MyStackWrapperInt =
            new MyStackWrapper<int>(stackInt);
    }
}
```

Compiler Warning (level 2) CS1711

Article • 09/15/2021

XML comment on 'type' has a typeparam tag for 'parameter', but there is no type parameter by that name

The documentation of a generic type includes a tag for the type parameter that has the wrong name.

Example

The following code generates warning CS1711.

C#

```
// cs1711.cs
// compile with: /doc:cs1711.xml
// CS1711 expected
using System;
///<typeparam name="WrongName">can be an int</typeparam>
class CMain
{
    public static void Main() { }
}
```

Compiler Warning (Level 2) CS1927

Article • 09/15/2021

Ignoring /win32manifest for module because it only applies to assemblies.

A win32 manifest is only applied at the assembly level. Your module will compile but it will not have a manifest.

To correct this error

1. Remove the **/win32manifest option**.

2. Compile the code as an assembly.

Example

The following example generates CS1927 when it is compiled with both the **/target:module** and **/win32manifest** compiler options.

C#

```
// cs1927.cs
// Compile with: /target:module /win32manifest
using System;

class ManifestWithModule
{
    static int Main()
    {
        return 1;
    }
}
```

See also

- **Win32Manifest** (C# Compiler Options)
- **TargetType** module (C# Compiler Options)

Compiler Warning (level 2) CS3019

Article • 09/15/2021

CLS compliance checking will not be performed on 'type' because it is not visible from outside this assembly.

This warning occurs when a type or a member that has the CLSCompliantAttribute attribute is not visible from another assembly. To resolve this error, remove the attribute on any classes or members that are not visible from the other assembly, or make the type or members visible. For more information on CLS compliance, see Language independence and language-independent components.

Example

The following sample generates CS3019:

C#

```
// CS3019.cs
// compile with: /W:2

using System;

[assembly: CLSCompliant(true)]

// To fix the error, remove the next line
[CLSCompliant(true)]  // CS3019
class C
{
    [CLSCompliant(false)]  // CS3019
    void Foo()
    {
    }

    static void Main()
    {
    }
}
```

See also

- Language Independence and Language-Independent Components

Compiler Warning (level 2) CS3021

Article • 09/15/2021

'type' does not need a CLSCompliant attribute because the assembly does not have a CLSCompliant attribute

This warning occurs if `[CLSCompliant(false)]` appears on a class in an assembly which does not have an assembly-level CLSCompliant attribute set to true (that is, the line `[assembly: CLSCompliant(true)]`). Since the assembly is not declaring itself CLS compliant, there is no need for anything within the assembly to declare itself non-compliant, since it is assumed to be non-compliant. For more information on CLS compliance, see Language independence and language-independent components.

To get rid of this warning, remove the attribute or add the assembly level attribute.

Example

The following example generates CS3021:

C#

```
// CS3021.cs
using System;
// Uncomment the following line to declare the assembly CLS Compliant,
// and avoid the warning without removing the attribute on the class.
//[assembly: CLSCompliant(true)]

// Remove the next line to avoid the warning.
[CLSCompliant(false)]               // CS3021
public class C
{
    public static void Main()
    {
    }
}
```

See also

- Language Independence and Language-Independent Components

Compiler Warning (level 3) CS0067

07/22/2025

The event 'event' is never used

An event was declared but never used in the class in which it was declared.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

The following sample generates CS0067:

C#

```
// CS0067.cs
// compile with: /W:3
using System;
delegate void MyDelegate();

class MyClass
{
   public event MyDelegate evt;   // CS0067
   // uncomment TestMethod to resolve this CS0067
/*
   private void TestMethod()
   {
      if (evt != null)
         evt();
   }
*/
   public static void Main()
   {
   }
}
```

If the event is unused intentionally, for example, when it's part of an interface implementation, then you can avoid emitting an unnecessary field as follows:

C#

```
using System;

public interface IThing
```

```
{
    event Action? E;
}

public class Thing : IThing
{
    // no CS0067 though the event is left unused
    public event Action? E { add { } remove { } }
}
```

Compiler Warning (level 3) CS0168

Article • 09/15/2021

The variable 'var' is declared but never used

The compiler issues a level-three warning when you declare a variable, but do not use it.

The following sample generates a CS0168 warning:

C#

```
// CS0168.cs
// compile with: /W:3
public class clx
{
    public int i;
}

public class clz
{
    public static void Main() {
        clx a; // CS0168, the variable 'a' is declared but never used
        // try the following line instead
        // clx a = new clx();  // this does not generate a warning because
the clx constructor must execute.
    }
}
```

Compiler Warning (level 3) CS0169

07/22/2025

The private field 'class member' is never used

A private variable was declared but never referenced. A common way to generate this warning is when you declare a private member of a class and do not use it.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

The following sample generates CS0169:

C#

```
// compile with: /W:3
using System;
public class ClassX
{
   int i;   // CS0169, i is not used anywhere
   // Remove the above variable declaration or uncomment TestMethod to clear
warning CS0169
   /*
   public void TestMethod()
   {
       i = 5;
       System.Console.WriteLine(i);
   }
   */

   public static void Main()
   {
   }
}
```

Compiler Warning (level 3) CS0219

Article • 09/15/2021

The variable 'variable' is assigned but its value is never used

The compiler issues a level-three warning, when you declare and assign a variable, but do not use it.

> **Note**
>
> The compiler generates this warning only when the variable value is a compile-time constant. Assigning a non-constant expression or method result to a local variable makes it easier to observe those expressions in the debugger. It also makes the result reachable, preventing garbage collection while that variable is reachable.

The following sample shows the cases when and when not the warning is generated:

C#

```
// CS0219.cs
// compile with: /W:3
public class MyClass
{
   public static void Main()
   {
      var interpolated = "Interpolated";
      var a = 0; // CS0219
      int b = GetZero(); // Doesn't generate a warning.
      var c = "Regular string"; // CS0219
      var d = $"Constant interpolated string"; // Doesn't generate a
warning.
      var e = $"{interpolated} string"; // Doesn't generate a warning.
   }

   private static int GetZero()
   {
       return 0;
   }
}
```

Compiler Warning (level 3) CS0414

07/22/2025

The private field 'field' is assigned but its value is never used

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

This warning can occur in several scenarios in which the compiler can verify that a variable is never referenced:

- A private field is assigned a constant value but is never subsequently read. The unnecessary assignment could effect performance. Consider removing the field.
- A private or internal static field is assigned a constant value only in the initializer. Consider changing the field to a const.
- A private or internal field is assigned constant values and only used in blocks that are excluded by #ifdef directives. Consider putting the field inside the #ifdef block.
- A private or internal field is assigned constant values in multiple locations but not otherwise accessed. If you do not need the field, consider removing it. Otherwise, use it in some appropriate way.

In other situations, or where the suggested workaround is not acceptable, use #pragma 0414.

The following sample shows one way in which CS0414 will be generated:

C#

```
// CS0414
// compile with: /W3
class C
{
   private int i = 1;  // CS0414

   public static void Main()
   { }
}
```

> **Note**
>
> If the variable `i` is declared as `protected` or `public`, no error will be generated because the compiler cannot know whether a derived class might use it or some other client code might instantiate the class and reference the variable

See also

- C# Compiler Errors
- C# Compiler Options

Compiler Warning (level 3) CS0419

Article • 09/15/2021

Ambiguous reference in cref attribute: 'Method Name1'. Assuming 'Method Name2', but could have also matched other overloads including 'Method Name3'.

In an XML documentation comment in the code, a reference could not be resolved. This could occur if the method is overloaded, or if two different identifiers with the same name are found. To resolve the warning, use a qualified name to disambiguate the reference, or include the specific overload in parentheses.

The following sample generates CS0419.

C#

```
// cs0419.cs
// compile with: /doc:x.xml /W:3
interface I
{
   /// text for F(void)
   void F();
   /// text for F(int)
   void F(int i);
}
/// text for class MyClass
public class MyClass
{
   /// <see cref="I.F"/>
   public static void MyMethod(int i)
   {
   }
/* Try this instead:
   /// <see cref="I.F(int)"/>
   public static void MyMethod(int i)
   {
   }
*/
   /// text for Main
   public static void Main ()
   {
   }
}
```

Compiler Warning (level 3) CS0642

Article • 05/13/2022

Possible mistaken empty statement

A semicolon after a conditional statement may cause your code to execute differently than intended.

You can use **NoWarn** compiler option or `#pragmas warning` to disable this warning; see **NoWarn** (C# Compiler Options) or #pragma warning for more information.

The following sample generates CS0642:

C#

```
// CS0642.cs
// compile with: /W:3
class MyClass
{
   public static void Main()
   {
      int i;

      for (i = 0; i < 10; i += 1);   // CS0642 semicolon intentional?
      {
         System.Console.WriteLine (i);
      }
   }
}
```

Compiler Warning (level 3) CS0659

Article • 09/15/2021

'class' overrides Object.Equals(object o) but does not override Object.GetHashCode()

The compiler detected an override of the Object.Equals method but no override of the Object.GetHashCode method. An override of Equals implies that you also want to override GetHashCode.

The following code generates CS0659:

C#

```
// CS0659.cs
// compile with: /W:3 /target:library
class Test
{
   public override bool Equals(object o) { return true; }   // CS0659
}

// OK
class Test2
{
   public override bool Equals(object o) { return true; }
   public override int GetHashCode() { return 0; }
}
```

See also

- Object.Equals
- Object.GetHashCode
- Equality Operators
- Implementing the Equals Method

Compiler Warning (level 3) CS0665

Article • 09/15/2021

Assignment in conditional expression is always constant; did you mean to use == instead of = ?

A conditional expression used the = operator and not the == operator.

The following sample generates CS0665:

C#

```
// CS0665.cs
// compile with: /W:3
class Test
{
   public static void Main()
   {
      bool i = false;

      if (i = true)   // CS0665
      // try the following line instead
      // if (i == true)
      {
      }

      System.Console.WriteLine(i);
   }
}
```

Compiler Warning (level 3) CS0675

Article • 10/27/2021

Bitwise-or operator used on a sign-extended operand; consider casting to a smaller unsigned type first

The compiler implicitly widened and sign-extended a variable, and then used the resulting value in a bitwise OR operation. This can result in unexpected behavior.

The following sample generates CS0675:

C#

```
// CS0675.cs
// compile with: /W:3
using System;

public class sign
{
   public static void Main()
   {
      int hi = 1;
      int lo = -1;
      long value = (((long)hi) << 32) | lo;              // CS0675, value
contains -1 (0xffffffff_ffffffff)
      // try the following line instead
      // long value = (((long)hi) << 32) | ((uint)lo);   // correct, value
contains 8589934591 (0x00000001_ffffffff)
   }
}
```

Compiler Warning (level 3) CS0693

Article • 09/15/2021

Type parameter 'type parameter' has the same name as the type parameter from outer type 'type'

This error occurs when you have a generic member such as a method inside a generic class. Since the method's type parameter is not necessarily the same as the class's type parameter, you cannot give them both the same name. For more information, see Generic Methods.

To avoid this situation, use a different name for one of the type parameters.

Example

The following sample generates CS0693.

C#

```
// CS0693.cs
// compile with: /W:3 /target:library
class Outer<T>
{
   class Inner<T> {}   // CS0693
   // try the following line instead
   // class Inner<U> {}
}
```

Compiler Warning (level 3) CS1700

Article • 09/15/2021

Assembly reference Assembly Name is invalid and cannot be resolved

This warning indicates that an attribute, such as InternalsVisibleToAttribute, was not specified correctly.

For more information, see Friend Assemblies.

Example

The following sample generates CS1700.

C#

```
// CS1700.cs
// compile with: /target:library
using System.Runtime.CompilerServices;
[assembly:InternalsVisibleTo("app2, Retargetable=f")]   // CS1700
[assembly:InternalsVisibleTo("app2")]   // OK
```

Compiler Warning (level 3) CS1702

09/02/2025

Assuming assembly reference "Assembly Name #1" matches "Assembly Name #2", you may need to supply runtime policy

The two assembly references have differing build and/or revision numbers, so will not automatically unify. You may need to supply run-time policy to force unification by `using` directives in the application .config file.

See also

- Configuration File Schema

Compiler Warning (level 3) CS1717

Article • 09/15/2021

Assignment made to same variable; did you mean to assign something else?

This warning occurs when you assign a variable to itself, such as `a = a`.

Several common mistakes can generate this warning:

- Writing `a = a` as the condition of an **if** statement, such as `if (a = a)`. You probably meant to say `if (a == a)`, which is always true, so you could write this more concisely as `if (true)`.
- Mistyping. You probably meant to say `a = b`.
- In a constructor where the parameter has the same name as the field, not using the **this** keyword: you probably meant to say `this.a = a`.

Example

The following sample generates CS1717.

C#

```
// CS1717.cs
// compile with: /W:3
public class Test
{
   public static void Main()
   {
      int x = 0;
      x = x;   // CS1717
   }
}
```

Compiler Warning (level 3) CS1718

Article • 09/15/2021

Comparison made to same variable; did you mean to compare something else?

If you meant to compare to something else, then correct the statement.

But another possibility is that you were testing for true or false, and were doing so by statements such as `if (a == a) (true)` or `if (a < a) (false)`. It is better to use `if (true)` or `if (false)`, for the following two reasons:

- It is simpler: it is always clearer to simply say what you mean.
- It helps avoid confusion: a new feature of C# 2.0 is nullable value types, which are analogous to the value `null` in T-SQL, the programming language used by SQL Server. Developers familiar with T-SQL might be concerned about the effect of nullable value types on expressions such as `if (a == a)`, because of the use of ternary logic in T-SQL. If you use `true` or `false`, you avoid this possible confusion.

Example

The following code generates warning CS1718.

C#

```
// CS1718.cs
using System;
public class Tester
{
    public static void Main()
    {
        int i = 0;
        if (i == i) { // CS1718.cs
        //if (true) {
            i++;
        }
    }
}
```

Compiler Warning (level 4) CS0028

07/22/2025

'function declaration' has the wrong signature to be an entry point

The method declaration for `Main` was invalid: it was declared with an invalid signature. `Main` must be declared as static and it must return either int or void. For more information, see Main() and Command-Line Arguments.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

The following sample generates CS0028:

C#

```
// CS0028.cs
// compile with: /W:4 /warnaserror
public class a
{
    public static double Main(int i)   // CS0028
    // Try the following line instead:
    // public static void Main()
    {
    }
}
```

Compiler Warning (level 4) CS0078

Article • 09/15/2021

The 'l' suffix is easily confused with the digit '1' -- use 'L' for clarity

The compiler warns when it detects a long literal using a lowercase l instead of an uppercase L.

The following sample generates CS0078:

C#

```
class Program
{
   public static void TestL(long i)
   {
   }

   public static void Main()
   {
      TestL(25l);   // CS0078
      // try the following line instead
      // TestL(25L);
   }
}
```

Compiler Warning (level 4) CS0109

Article • 09/15/2021

The member 'member' does not hide an inherited member. The new keyword is not required

A class declaration included the new keyword even though the declaration does not override an existing declaration in a base class. You can delete the **new** keyword.

The following sample generates CS0109:

C#

```
// CS0109.cs
// compile with: /W:4
namespace x
{
   public class a
   {
      public int i;
   }

   public class b : a
   {
      public new int i;
      public new int j;   // CS0109
      public static void Main()
      {
      }
   }
}
```

Compiler Warning (level 4) CS0402

07/22/2025

'identifier' : an entry point cannot be generic or in a generic type

The entry point was found in a generic type. To remove this warning, implement Main in a non-generic class or struct.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

C#

```
// CS0402.cs
// compile with: /W:4
class C<T>
{
   public static void Main()  // CS0402
   {

   }
}

class CMain
{
   public static void Main() {}
}
```

Compiler Warning (level 4) CS0422

Article • 09/15/2021

The /incremental option is no longer supported

Incremental compilation (**/incr** or **/incremental**) is not supported in C# 2.0.

> **Note**
>
> This compiler warning is no longer used in Roslyn.

Compiler Warning (level 4) CS0429

Article • 09/15/2021

Unreachable expression code detected

This error occurs whenever part of an expression in your code is unreachable. In the following example, the condition `false && myTest()` meets this criteria because the `myTest()` method will never get evaluated due to the fact that the left side of the `&&` operation is always false. As soon as the `&&` operator evaluates the `false` statement as false, it stops the evaluation, and will never evaluate the right side.

Example

The following code generates CS0429.

C#

```
// CS0429.cs
public class cs0429
{
    public static void Main()
    {
        if (false && myTest())  // CS0429
        // Try the following line instead:
        // if (true && myTest())
        {
        }
        else
        {
            int i = 0;
            i++;
        }
    }

    static bool myTest() { return true; }
}
```

Compiler Warning (level 4) CS0628

Article • 09/15/2021

'member' : new protected member declared in sealed class

A sealed class cannot introduce a protected member because no other class will be able to inherit from the `sealed` class and use the `protected` member.

The following sample generates CS0628:

C#

```
// CS0628.cs
// compile with: /W:4
sealed class C
{
   protected int i;   // CS0628
}

class MyClass
{
   public static void Main()
   {
   }
}
```

Compiler Warning (level 4) CS0649

07/22/2025

Field 'field' is never assigned to, and will always have its default value 'value'

The compiler detected an uninitialized private or internal field declaration that is never assigned a value.

> **Note**
>
> This warning is only reported during explicit **Build** or **Rebuild** operations. It does not appear during typing in the IDE as part of IntelliSense diagnostics. This means that if you fix the warning by using the field or removing it, the warning might persist in the error list until you build or rebuild the project again.

The following sample generates CS0649:

C#

```
// CS0649.cs
// compile with: /W:4
using System.Collections;

class MyClass
{
   Hashtable table;  // CS0649
   // You may have intended to initialize the variable to null
   // Hashtable table = null;

   // Or you may have meant to create an object here
   // Hashtable table = new Hashtable();

   public void Func(object o, string p)
   {
      // Or here
      // table = new Hashtable();
      table[p] = o;
   }

   public static void Main()
   {
   }
}
```

Compiler Warning (level 4) CS1573

Article • 09/15/2021

Parameter 'parameter' has no matching param tag in the XML comment for 'parameter' (but other parameters do)

When using the **DocumentationFile** compiler option, a comment was specified for some but not all parameters in a method. You may have forgotten to enter a comment for these parameters.

The following sample generates CS1573:

C#

```
// CS1573.cs
// compile with: /W:4
public class MyClass
{
   /// <param name='Int1'>Used to indicate status.</param>
   // enter a comment for Char1?
   public static void MyMethod(int Int1, char Char1)
   {
   }

   public static void Main ()
   {
   }
}
```

Compiler Warning (level 4) CS1591

Article • 12/14/2022

Missing XML comment for publicly visible type or member 'Type_or_Member'

The **DocumentationFile** compiler option was specified, but one or more constructs did not have comments.

The following sample generates CS1591:

C#

```
// CS1591.cs
// compile with: /W:4 /doc:x.xml

/// text
public class Test
{
   // /// text
   public static void Main()   // Try the following instead: remove "//"
from previous line (`// /// text` about the comments)
                               //or add #pragma warning disable 1591
                               //a good practice is add the following
comments:
                               // /// <summary>
                               // ///
                               // /// </summary>
                               // this can be easily achieve at Visual
Studio in a simple way adding triple forward slashes (`///`)
   {
   }
}
```

Compiler Warning (level 4) CS1610

Article • 09/15/2021

Unable to delete temporary file 'file' used for default Win32 resource -- resource

When using the **Win32Resource** compiler option and when your **%TEMP%** directory does not have DELETE permission, this warning indicates that the compiler could not delete a temporary file that it created.

Make sure that you have read/write/delete permissions for the **%TEMP%** directory.

If necessary, you can manually delete these files and there will be no harm to C# or any of your programs.

Compiler Warning (level 4) CS1712

Article • 09/15/2021

Type parameter 'type parameter' has no matching typeparam tag in the XML comment on 'type' (but other type parameters do)

The documentation of a generic type is missing a **typeparam** tag. For more information, see <typeparam>.

Example

The following code generates warning CS1712. To resolve this error, add a **typeparam** tag for the type parameter S.

C#

```
// CS1712.cs
// compile with: /doc:cs1712.xml
using System;
class Test
{
    public static void Main() {}
    /// <param name="j"> This is the j parameter.</param>
    /// <typeparam name="T"> This is the T type parameter.</typeparam>
    public void bar<T,S>(int j) {}  // CS1712
}
```

C# documentation

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

Learn to program

GET STARTED

Learn C# | Tutorials, courses, videos, and more

Beginner C# tutorials in your browser

VIDEO

C# beginner video series

TUTORIAL

Self-guided tutorials

In-browser tutorial

REFERENCE

C# on Q&A

Languages on .NET tech community forums

C# on Stack Overflow

C# on Discord

Fundamentals

OVERVIEW

A tour of C#

Inside a C# program

C# highlights video series

C# Language development strategy

CONCEPT

Type system

Object oriented programming

Functional techniques

Exceptions

Coding style

TUTORIAL

Display command-line

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

What's new in C# 11

TUTORIAL

Explore record types

Explore top level statements

REFERENCE

Breaking changes in the C# compiler

Version compatibility

Key concepts

OVERVIEW

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

Advanced concepts

REFERENCE

Reflection and attributes

Expression trees

Native interoperability

Performance engineering

.NET Compiler Platform SDK

C# language reference

REFERENCE

Language reference

Keywords

Operators and expressions

Tokens

C# language standard

REFERENCE

Overview

C# language specification

Feature specifications

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