What's new in C# 14

C# 14 includes the following new features. You can try these features using the latest Visual Studio 2022 version or the .NET 10 SDK :

- Extension members
- Null-conditional assignment
- nameof supports unbound generic types
- More implicit conversions for Span<T> and ReadOnlySpan<T>
- Modifiers on simple lambda parameters
- field backed properties
- partial events and constructors
- user-defined compound assignment operators

C# 14 is supported on **.NET 10**. For more information, see C# language versioning.

You can download the latest .NET 10 SDK from the .NET downloads page . You can also download Visual Studio 2022 , which includes the .NET 10 SDK.

New features are added to the "What's new in C#" page when they're available in public preview releases. The working set section of the roslyn feature status page tracks when upcoming features are merged into the main branch.

You can find any breaking changes introduced in C# 14 in our article on breaking changes.

> **Note**
>
> We're interested in your feedback on these features. If you find issues with any of these new features, create a **new issue** in the **dotnet/roslyn** repository.

Extension members

C# 14 adds new syntax to define *extension members*. The new syntax enables you to declare *extension properties* in addition to extension methods. You can also declare extension members that extend the type, rather than an instance of the type. In other words, these new extension members can appear as static members of the type you extend. These extensions can include user defined operators implemented as static extension methods. The following code example shows an example of the different kinds of extension members you can declare:

C#

```
public static class Enumerable
{
    // Extension block
    extension<TSource>(IEnumerable<TSource> source) // extension members for
IEnumerable<TSource>
    {
        // Extension property:
        public bool IsEmpty => !source.Any();

        // Extension method:
        public IEnumerable<TSource> Where(Func<TSource, bool> predicate) {
... }
    }

    // extension block, with a receiver type only
    extension<TSource>(IEnumerable<TSource>) // static extension members for
IEnumerable<Source>
    {
        // static extension method:
        public static IEnumerable<TSource> Combine(IEnumerable<TSource>
first, IEnumerable<TSource> second) { ... }

        // static extension property:
        public static IEnumerable<TSource> Identity =>
Enumerable.Empty<TSource>();

        // static user defined operator:
        public static IEnumerable<TSource> operator + (IEnumerable<TSource>
left, IEnumerable<TSource> right) => left.Concat(right);
    }
}
```

The members in the first extension block are called as though they're instance members of `IEnumerable<TSource>`, for example `sequence.IsEmpty`. The members in the second extension block are called as though they're static members of `IEnumerable<TSource>`, for example `IEnumerable<int>.Identity`.

You can learn more details by reading the article on extension members in the programming guide, the language reference article on the extension keyword, and the feature specification for the new extension members feature.

The `field` keyword

The token `field` enables you to write a property accessor body without declaring an explicit backing field. The token `field` is replaced with a compiler synthesized backing field.

For example, previously, if you wanted to ensure that a `string` property couldn't be set to `null`, you had to declare a backing field and implement both accessors:

```C#
private string _msg;
public string Message
{
    get => _msg;
    set => _msg = value ?? throw new ArgumentNullException(nameof(value));
}
```

You can now simplify your code to:

```C#
public string Message
{
    get;
    set => field = value ?? throw new ArgumentNullException(nameof(value));
}
```

You can declare a body for one or both accessors for a field backed property.

There's a potential breaking change or confusion reading code in types that also include a symbol named `field`. You can use `@field` or `this.field` to disambiguate between the `field` keyword and the identifier, or you can rename the current `field` symbol to provide better distinction.

If you try this feature and have feedback, comment on the feature issue in the `csharplang` repository.

The field contextual keyword is in C# 13 as a preview feature.

Implicit span conversions

C# 14 introduces first-class support for System.Span<T> and System.ReadOnlySpan<T> in the language. This support involves new implicit conversions allowing more natural programming with these types.

`Span<T>` and `ReadOnlySpan<T>` are used in many key ways in C# and the runtime. Their introduction improves performance without risking safety. C# 14 recognizes the relationship and supports some conversions between `ReadOnlySpan<T>`, `Span<T>`, and `T[]`. The span types can be extension method receivers, compose with other conversions, and help with generic type inference scenarios.

You can find the list of implicit span conversions in the article on built-in types in the language reference section. You can learn more details by reading the feature specification for First class

span types.

Unbound generic types and `nameof`

Beginning with C# 14, the argument to `nameof` can be an unbound generic type. For example, `nameof(List<>)` evaluates to `List`. In earlier versions of C#, only closed generic types, such as `List<int>`, could be used to return the `List` name.

Simple lambda parameters with modifiers

You can add parameter modifiers, such as `scoped`, `ref`, `in`, `out`, or `ref readonly` to lambda expression parameters without specifying the parameter type:

C#

```
delegate bool TryParse<T>(string text, out T result);
// ...
TryParse<int> parse1 = (text, out result) => Int32.TryParse(text, out
result);
```

Previously, adding any modifiers was allowed only when the parameter declarations included the types for the parameters. The preceding declaration would require types on all parameters:

C#

```
TryParse<int> parse2 = (string text, out int result) => Int32.TryParse(text,
out result);
```

The `params` modifier still requires an explicitly typed parameter list.

You can read more about these changes in the article on lambda expressions in the C# language reference.

More partial members

You can now declare instance constructors and events as partial members.

Partial constructors and partial events must include exactly one *defining declaration* and one *implementing declaration*.

Only the implementing declaration of a partial constructor can include a constructor initializer: `this()` or `base()`. Only one partial type declaration can include the primary constructor

syntax.

The implementing declaration of a partial event must include `add` and `remove` accessors. The defining declaration declares a field-like event.

User defined compound assignment

You can learn more in the feature specification for user-defined compound assignment.

Null-conditional assignment

The null-conditional member access operators, `?.` and `?[]`, can now be used on the left hand side of an assignment or compound assignment.

Before C# 14, you needed to null-check a variable before assigning to a property:

```
C#
if (customer is not null)
{
    customer.Order = GetCurrentOrder();
}
```

You can simplify the preceding code using the `?.` operator:

```
C#
customer?.Order = GetCurrentOrder();
```

The right side of the `=` operator is evaluated only when the left side isn't null. If `customer` is null, the code doesn't call `GetCurrentOrder`.

In addition to assignment, you can use null-conditional member access operators with compound assignment operators (`+=`, `-=`, and others). However, increment and decrement, `++` and `--`, aren't allowed.

You can learn more in the language reference article on the conditional member access and the feature specification for null-conditional assignment.

See also

- What's new in .NET 10

Last updated on 09/17/2025