

What's new in C# 9.0

09/04/2020 • 16 minutes to read •     

In this article

[Record types](#)

[Init only setters](#)

[Top-level statements](#)

[Pattern matching enhancements](#)

[Performance and interop](#)

[Fit and finish features](#)

[Support for code generators](#)

C# 9.0 adds the following features and enhancements to the C# language:

- [Records](#)
- [Init only setters](#)
- [Top-level statements](#)
- [Pattern matching enhancements](#)
- Native sized integers
- Function pointers
- Suppress emitting localsinit flag
- Target-typed new expressions
- static anonymous functions
- Target-typed conditional expressions
- Covariant return types
- Extension GetEnumerator support for foreach loops
- Lambda discard parameters
- Attributes on local functions
- Module initializers
- New features for partial methods

C# 9.0 is supported on **.NET 5**. For more information, see [C# language versioning](#).


Record types

C# 9.0 introduces **record types**, which are a reference type that provides synthesized methods to provide value semantics for equality. Records are immutable by default.

Record types make it easy to create immutable reference types in .NET. Historically, .NET types are largely classified as reference types (including classes and anonymous types)

and value types (including structs and tuples). While immutable value types are recommended, mutable value types don't often introduce errors. Value type variables hold the values so changes are made to a copy of the original data when value types are passed to methods.


There are many advantages to immutable reference types as well. These advantages are more pronounced in concurrent programs with shared data. Unfortunately, C# forced you to write quite a bit of extra code to create immutable reference types. Records provide a type declaration for an immutable reference type that uses value semantics for equality. The synthesized methods for equality and hash codes consider two records equal if their properties are all equal. Consider this definition:

| | |
|---|--|
| C# |  Copy |
| <pre>public record Person { public string LastName { get; } public string FirstName { get; } public Person(string first, string last) => (FirstName, LastName) = (first, last); }</pre> | |


The record definition creates a `Person` type that contains two readonly properties: `FirstName` and `LastName`. The `Person` type is a reference type. If you looked at the IL, it's a class. It's immutable in that none of the properties can be modified once it's been created. When you define a record type, the compiler synthesizes several other methods for you:

- Methods for value-based equality comparisons
- Override for `GetHashCode()`
- Copy and Clone members
- `PrintMembers` and `ToString()`

Records support inheritance. You can declare a new record derived from `Person` as follows:

| | |
|--|--|
| C# |  Copy |
| <pre>public record Teacher : Person { public string Subject { get; } public Teacher(string first, string last, string sub) : base(first, last) => Subject = sub; }</pre> | |

You can also seal records to prevent further derivation:

| C# |  Copy |
|---|--|
| <pre>public sealed record Student : Person { public int Level { get; } public Student(string first, string last, int level) : base(first, last) => Level = level; }</pre> | |

The compiler synthesizes different versions of the methods above. The method signatures depend on if the record type is sealed and if the direct base class is object. Records should have the following capabilities:

- Equality is value-based, and includes a check that the types match. For example, a `Student` can't be equal to a `Person`, even if the two records share the same name.
- Records have a consistent string representation generated for you.
- Records support copy construction. Correct copy construction must include inheritance hierarchies, and properties added by developers.
- Records can be copied with modification. These copy and modify operations supports non-destructive mutation.


In addition to the familiar `Equals` overloads, operator `==`, and operator `!=`, the compiler synthesizes a new `EqualityContract` property. The property returns a `Type` object that matches the type of the record. If the base type is `object`, the property is `virtual`. If the base type is another record type, the property is an `override`. If the record type is `sealed`, the property is `sealed`. The synthesized `GetHashCode` uses the `GetHashCode` from all properties and fields declared in the base type and the record type. These synthesized methods enforce value-based equality throughout an inheritance hierarchy. That means a `Student` will never be considered equal to a `Person` with the same name. The types of the two records must match as well as all properties shared among the record types being equal.

Records also have a synthesized constructor and a "clone" method for creating copies. The synthesized constructor has one argument of the record type. It produces a new record with the same values for all properties of the record. This constructor is private if the record is sealed, otherwise it's protected. The synthesized "clone" method supports copy construction for record hierarchies. The term "clone" is in quotes because the actual name is compiler generated. You can't create a method named `Clone` in a record type. The synthesized "clone" method returns the type of record being copied using


virtual dispatch. The compiler adds different modifiers for the "clone" method depending on the access modifiers on the record:

- If the record type is `abstract`, the "clone" method is also `abstract`. If the base type isn't `object`, the method is also `override`.
- For record types that aren't `abstract` when the base type is `object`:
 - If the record is `sealed`, no additional modifiers are added to the "clone" method (meaning it is not `virtual`).
 - If the record isn't `sealed`, the "clone" method is `virtual`.
- For record types that aren't `abstract` when the base type is not `object`:
 - If the record is `sealed`, the "clone" method is also `sealed`.
 - If the record isn't `sealed`, the "clone" method is `override`.


The result of all these rules is the equality is implemented consistently across any hierarchy of record types. Two records are equal to each other if their properties are equal and their types are the same, as shown in the following example:

| | |
|--|--|
| C# |  Copy |
| <pre>var person = new Person("Bill", "Wagner"); var student = new Student("Bill", "Wagner", 11); Console.WriteLine(student == person); // false</pre> | |

The compiler synthesizes two methods that support printed output: a `ToString()` override, and `PrintMembers`. The `PrintMembers` takes a `System.Text.StringBuilder` as its argument. It appends a comma-separated list of property names and values for all properties in the record type. `PrintMembers` calls the base implementation for any records derived from other records. The `ToString()` override returns the string produced by `PrintMembers`, surrounded by `{` and `}`. For example, the `ToString()` method for `Student` returns a string like the following code:

| | |
|--|--|
| C# |  Copy |
| <pre>"Student { LastName = Wagner, FirstName = Bill, Level = 11 }"</pre> | |

The examples shown so far use traditional syntax to declare properties. There's a more concise form called ***positional records***. Here are the three record types defined earlier as positional records:

| | |
|----|--|
| C# |  Copy |
|----|--|

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

public record Teacher(string FirstName, string LastName,
    string Subject)
    : Person(FirstName, LastName);

public sealed record Student(string FirstName,
    string LastName, int Level)
    : Person(FirstName, LastName);
```

These declarations create the same functionality as the earlier version (with a couple extra features covered in the following section). These declarations end with a semicolon instead of brackets because these records don't add additional methods. You can add a body, and include any additional methods as well:

C#

 Copy

```
public record Pet(string Name)
{
    public void ShredTheFurniture() =>
        Console.WriteLine("Shredding furniture");
}

public record Dog(string Name) : Pet(Name)
{
    public void WagTail() =>
        Console.WriteLine("It's tail wagging time");

    public override string ToString()
    {
        StringBuilder s = new();
        base.PrintMembers(s);
        return $"{s.ToString()} is a dog";
    }
}
```

The compiler produces a `Deconstruct` method for positional records. The `Deconstruct` method has parameters that match the names of all public properties in the record type. The `Deconstruct` method can be used to deconstruct the record into its component properties:


C#

 Copy

```
var person = new Person("Bill", "Wagner");

var (first, last) = person;
Console.WriteLine(first);
Console.WriteLine(last);
```

Finally, records support *with-expressions*. A *with-expression* instructs the compiler to create a copy of a record, but *with* specified properties modified:


| | |
|---|--|
| C# |  Copy |
| <pre>Person brother = person with { FirstName = "Paul" };</pre> | |

The above line creates a new `Person` record where the `LastName` property is a copy of `person`, and the `FirstName` is "Paul". You can set any number of properties in a *with-expression*. Any of the synthesized members except the "clone" method may be written by you. If a record type has a method that matches the signature of any synthesized method, the compiler doesn't synthesize that method. The earlier `Dog` record example contains a hand coded `ToString()` method as an example.


Init only setters

Init only setters provide consistent syntax to initialize members of an object. Property initializers make it clear which value is setting which property. The downside is that those properties must be settable. Starting with C# 9.0, you can create *init* accessors instead of *set* accessors for properties and indexers. Callers can use property initializer syntax to set these values in creation expressions, but those properties are readonly once construction has completed. *Init only setters* provide a window to change state. That window closes when the construction phase ends. The construction phase effectively ends after all initialization, including property initializers and *with-expressions* have completed.


The preceding example for positional records demonstrates using an *init-only* setter to set a property using a *with* expression. You can declare *init only* setters in any type you write. For example, the following struct defines a weather observation structure:

| | |
|--|--|
| C# |  Copy |
| <pre>public struct WeatherObservation { public DateTime RecordedAt { get; init; } public decimal TemperatureInCelsius { get; init; } public decimal PressureInMillibars { get; init; } public override string ToString() => \$"At {RecordedAt:h:mm tt} on {RecordedAt:M/d/yyyy}: " + \$"Temp = {TemperatureInCelsius}, with {PressureInMillibars} pressure"; }</pre> | |

Callers can use property initializer syntax to set the values, while still preserving the immutability:

| C# |  Copy |
|---|--|
| <pre>var now = new WeatherObservation { RecordedAt = DateTime.Now, TemperatureInCelsius = 20, PressureInMillibars = 998.0m };</pre> | |

But, changing an observation after initialization is an error by assigning to an init-only property outside of initialization:

| C# |  Copy |
|---|--|
| <pre>// Error! CS8852. now.TemperatureInCelsius = 18;</pre> | |


Init only setters can be useful to set base class properties from derived classes. They can also set derived properties through helpers in a base class. Positional records declare properties using init only setters. Those setters are used in with-expressions. You can declare init only setters for any `class` or `struct` you define.

Top-level statements

Top-level statements remove unnecessary ceremony from many applications. Consider the canonical "Hello World!" program:

| C# |  Copy |
|---|--|
| <pre>using System; namespace HelloWorld { class Program { static void Main(string[] args) { Console.WriteLine("Hello World!"); } } }</pre> | |

There's only one line of code that does anything. With top-level statements, you can replace all that boilerplate with the `using` statement and the single line that does the work:

| | |
|--|--|
| C# |  Copy |
| <pre>using System; Console.WriteLine("Hello World!");</pre> | |

If you wanted a one-line program, you could remove the `using` directive and use the fully qualified type name:

| | |
|--|--|
| C# |  Copy |
| <pre>System.Console.WriteLine("Hello World!");</pre> | |

Only one file in your application may use top-level statements. If the compiler finds top-level statements in multiple source files, it's an error. It's also an error if you combine top-level statements with a declared program entry point method, typically a `Main` method. In a sense, you can think that one file contains the statements that would normally be in the `Main` method of a `Program` class.

One of the most common uses for this feature is creating teaching materials. Beginner C# developers can write the canonical "Hello World!" in one or two lines of code. None of the extra ceremony is needed. However, seasoned developers will find many uses for this feature as well. Top-level statements enable a script-like experience for experimentation similar to what Jupyter notebooks provide. Top-level statements are great for small console programs and utilities. Azure functions are an ideal use case for top-level statements.


Most importantly, top-level statements don't limit your application's scope or complexity. Those statements can access or use any .NET class. They also don't limit your use of command-line arguments or return values. Top-level statements can access an array of strings named `args`. If the top-level statements return an integer value, that value becomes the integer return code from a synthesized `Main` method. The top-level statements may contain `async` expressions. In that case, the synthesized entry point returns a `Task`, or `Task<int>`.

Pattern matching enhancements


C# 9 includes new pattern matching improvements:

- **Type patterns** match a variable is a type
- **Parenthesized patterns** enforce or emphasize the precedence of pattern combinations
- **Conjunctive and patterns** require both patterns to match
- **Disjunctive or patterns** require either pattern to match
- **Negated not patterns** require that a pattern doesn't match
- **Relational patterns** require the input be less than, greater than, less than or equal, or greater than or equal to a given constant.


These patterns enrich the syntax for patterns. Consider these examples:

| C# |  Copy |
|--|--|
| <pre>public static bool IsLetter(this char c) => c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';</pre> | |

Alternatively, with optional parentheses to make it clear that `and` has higher precedence than `or`:

| C# |  Copy |
|---|---|
| <pre>public static bool IsLetterOrSeparator(this char c) => c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';</pre> | |

One of the most common uses is a new syntax for a null check:

| C# |  Copy |
|--|--|
| <pre>if (e is not null) { // ... }</pre> | |

Any of these patterns can be used in any context where patterns are allowed: `is` pattern expressions, `switch` expressions, nested patterns, and the pattern of a `switch` statement's `case` label.

Performance and interop

Three new features improve support for native interop and low-level libraries that require high performance: native sized integers, function pointers, and omitting the `localsinit` flag.

Native sized integers, `nint` and `nuint`, are integer types. They're expressed by the underlying types [System.IntPtr](#) and [System.UIntPtr](#). The compiler surfaces additional conversions and operations for these types as native ints. Native sized integers define properties for `MaxValue` or `MinValue`. These values can't be expressed as compile time constants because they depend on the native size of an integer on the target machine. Those values are readonly at runtime. You can use constant values for `nint` in the range `[int.MinValue .. int.MaxValue]`. You can use constant values for `nuint` in the range `[uint.MinValue .. uint.MaxValue]`. The compiler performs constant folding for all unary and binary operators using the [System.Int32](#) and [System.UInt32](#) types. If the result doesn't fit in 32 bits, the operation is executed at runtime and isn't considered a constant. Native sized integers can increase performance in scenarios where integer math is used extensively and needs to have the fastest performance possible.


Function pointers provide an easy syntax to access the IL opcodes `ldftn` and `calli`. You can declare function pointers using new `delegate*` syntax. A `delegate*` type is a pointer type. Invoking the `delegate*` type uses `calli`, in contrast to a delegate that uses `callvirt` on the `Invoke()` method. Syntactically, the invocations are identical. Function pointer invocation uses the `managed` calling convention. You add the `unmanaged` keyword after the `delegate*` syntax to declare that you want the `unmanaged` calling convention. Other calling conventions can be specified using attributes on the `delegate*` declaration.

Finally, you can add the [System.Runtime.CompilerServices.SkipLocalsInitAttribute](#) to instruct the compiler not to emit the `localsinit` flag. This flag instructs the CLR to zero-initialize all local variables. The `localsinit` flag has been the default behavior for C# since 1.0. However, the extra zero-initialization may have measurable performance impact in some scenarios. In particular, when you use `stackalloc`. In those cases, you can add the [SkipLocalsInitAttribute](#). You may add it to a single method or property, or to a `class`, `struct`, `interface`, or even a module. This attribute doesn't affect abstract methods; it affects the code generated for the implementation.


These features can improve performance in some scenarios. They should be used only after careful benchmarking both before and after adoption. Code involving native sized integers must be tested on multiple target platforms with different integer sizes. The other features require unsafe code.

Fit and finish features


Many of the other features help you write code more efficiently. In C# 9.0, you can omit the type in a [new expression](#) when the created object's type is already known. The most common use is in field declarations:

| | |
|--|---|
| C# |  Copy |
| <pre>private List<WeatherObservation> _observations = new();</pre> | |


Target-typed `new` can also be used when you need to create a new object to pass as an argument to a method. Consider a `ForecastFor()` method with the following signature:

| | |
|--|--|
| C# |  Copy |
| <pre>public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)</pre> | |

You could call it as follows:

| | |
|--|--|
| C# |  Copy |
| <pre>var forecast = station.ForecastFor(DateTime.Now.AddDays(2), new());</pre> | |

Another nice use for this feature is to combine it with init only properties to initialize a new object:

| | |
|---|--|
| C# |  Copy |
| <pre>WeatherStation station = new() { Location = "Seattle, WA" };</pre> | |

You can return an instance created by the default constructor using a `return new();` statement.

A similar feature improves the target type resolution of [conditional expressions](#). With this change, the two expressions need not have an implicit conversion from one to the other, but may both have implicit conversions to a target type. You likely won't notice this change. What you will notice is that some conditional expressions that previously required casts or wouldn't compile now just work.

Starting in C# 9.0, you can add the `static` modifier to [lambda expressions](#) or [anonymous methods](#). Static lambda expressions are analogous to the `static` local functions: a static lambda or anonymous method can't capture local variables or instance state. The `static` modifier prevents accidentally capturing other variables.

Covariant return types provide flexibility for the return types of overridden functions. An overridden virtual function can return a type derived from the return type declared in the base class method. This can be useful for Records, and for other types that support virtual clone or factory methods.

In addition, the [foreach loop](#) will recognize and use an extension method `GetEnumerator` that otherwise satisfies the `foreach` pattern. This change means `foreach` is consistent with other pattern-based constructions such as the `async` pattern, and pattern-based deconstruction. In practice, this change means you can add `foreach` support to any type. You should limit its use to when enumerating an object makes sense in your design.

Next, you can use `discards` as parameters to lambda expressions. This convenience enables you to avoid naming the argument, and the compiler may avoid using it. You use the `_` for any argument. For more information, see the [Input parameters of a lambda expression](#) section of the [Lambda expressions](#) article.

Finally, you can now apply attributes to [local functions](#). For example, you can apply [nullable attribute annotations](#) to local functions.

Support for code generators

Two final features support C# code generators. C# code generators are a component you can write that is similar to a Roslyn analyzer or code fix. The difference is that code generators analyze code and write new source code files as part of the compilation process. A typical code generator searches code for attributes or other conventions.

A code generator reads attributes or other code elements using the Roslyn analysis APIs. From that information, it adds new code to the compilation. Source generators can only add code; they aren't allowed to modify any existing code in the compilation.

The two features added for code generators are extensions to ***partial method syntax***, and ***module initializers***. First, the changes to partial methods. Before C# 9.0, partial methods are `private` but can't specify an access modifier, have a `void` return, and can't have out parameters. These restrictions meant that if no method implementation is provided, the compiler removes all calls to the partial method. C# 9.0 removes these restrictions, but requires that partial method declarations have an implementation. Code generators can provide that implementation. To avoid introducing a breaking change, the compiler considers any partial method without an access modifier to follow the old rules. If the partial method includes the `private` access modifier, the new rules govern that partial method.

The second new feature for code generators is ***module initializers***. Module initializers are methods that have the [ModuleInitializerAttribute](#) attribute attached to them. These methods will be called by the runtime when the assembly loads. A module initializer method:

- Must be static

- Must be parameterless
- Must return void
- Must not be a generic method
- Must not be contained in a generic class
- Must be accessible from the containing module

That last bullet point effectively means the method and its containing class must be internal or public. The method can't be a local function.

Is this page helpful?

 Yes  No
