-

-

-

-

-

-

**What's new in C# 9.0**

* Article
* 11/05/2021
* 17 minutes to read
* 9 contributors

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

* [Records](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#record-types)
* [Init only setters](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#init-only-setters)
* [Top-level statements](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#top-level-statements)
* [Pattern matching enhancements](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#pattern-matching-enhancements)
* [Performance and interop](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#performance-and-interop)
  + [Native sized integers](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/native-integers)
  + [Function pointers](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/function-pointers)
  + [Suppress emitting localsinit flag](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/skip-localsinit)
* [Fit and finish features](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#fit-and-finish-features)
  + [Target-typed new expressions](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/target-typed-new)
  + [static anonymous functions](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/static-anonymous-functions)
  + [Target-typed conditional expressions](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/target-typed-conditional-expression)
  + [Covariant return types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/covariant-returns)
  + [Extension GetEnumerator support for foreach loops](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/extension-getenumerator)
  + [Lambda discard parameters](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/lambda-discard-parameters)
  + [Attributes on local functions](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/local-function-attributes)
* [Support for code generators](https://docs.microsoft.com/en-us/dotnet/csharp/whats-new/csharp-9#support-for-code-generators)
  + [Module initializers](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/module-initializers)
  + [New features for partial methods](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/proposals/csharp-9.0/extending-partial-methods)

C# 9.0 is supported on **.NET 5**. For more information, see [C# language versioning](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version).

You can download the latest .NET SDK from the [.NET downloads page](https://dotnet.microsoft.com/download).

**Positional syntax for property definition**

You can use positional parameters to declare properties of a record and to initialize the property values when you create an instance:

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 init-only auto-implemented property for each positional parameter provided in the record declaration. An [init-only](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/init) property can only be set in the constructor or by using a property initializer.
* A primary constructor whose parameters match the positional parameters on the record declaration.
* A Deconstruct method with an out parameter for each positional parameter provided in the record declaration.

For more information, see [Positional syntax](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#positional-syntax-for-property-definition) in the C# language reference article about records.

**Immutability**

A record type is not necessarily immutable. You can declare properties with set accessors and fields that aren't readonly. But while records can be mutable, they make it easier to create immutable data models. Properties that you create by using positional syntax are immutable.

Immutability can be useful when you want a data-centric type to be thread-safe or a hash code to remain the same in a hash table. It can prevent bugs that happen when you pass an argument by reference to a method, and the method unexpectedly changes the argument value.

The features unique to record types are implemented by compiler-synthesized methods, and none of these methods compromises immutability by modifying object state.

**Value equality**

Value equality means that two variables of a record type are equal if the types match and all property and field values match. For other reference types, equality means identity. That is, two variables of a reference type are equal if they refer to the same object.

The following example illustrates value equality of record types:

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

}

In class types, you could manually override equality methods and operators to achieve value equality, but developing and testing that code would be time-consuming and error-prone. Having this functionality built-in prevents bugs that would result from forgetting to update custom override code when properties or fields are added or changed.

For more information, see [Value equality](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#value-equality) in the C# language reference article about records.

**Nondestructive mutation**

If you need to mutate immutable properties of a record instance, 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](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/object-and-collection-initializers) syntax to specify the values to be changed, as shown in the following example:

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

}

For more information, see [Nondestructive mutation](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#nondestructive-mutation) in the C# language reference article about records.

**Built-in formatting for display**

Record types have a compiler-generated [ToString](https://docs.microsoft.com/en-us/dotnet/api/system.object.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>, ...}

For reference types, the type name of the object that the property refers to is displayed instead of the property value. In the following example, the array is a reference type, so System.String[] is displayed instead of the actual array element values:

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

For more information, see [Built-in formatting](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#built-in-formatting-for-display) in the C# language reference article about records.

**Inheritance**

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.

The following example illustrates inheritance with positional property syntax:

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 }

}

For two record variables to be equal, the run-time type must be equal. The types of the containing variables might be different. This is illustrated in the following code example:

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 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.

All public properties and fields of both derived and base types are included in the ToString output, as shown in the following example:

C#Copy

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 }

}

For more information, see [Inheritance](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/record#inheritance) in the C# language reference article about records.

**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.

You can declare init only setters in any type you write. For example, the following struct defines a weather observation structure:

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";

}

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

var now = new WeatherObservation

{

RecordedAt = DateTime.Now,

TemperatureInCelsius = 20,

PressureInMillibars = 998.0m

};

An attempt to change an observation after initialization results in a compiler error:

// Error! CS8852.

now.TemperatureInCelsius = 18;

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, struct, or record you define.

For more information, see [init (C# Reference)](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/init).

**Top-level statements**

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

**using System;**

namespace HelloWorld

{

class Program

{

static void Main(string[] args)

{

Console.WriteLine("Hello World!");

}

}

}

-

**boilerplate** meaning:

a way of writing or thinking that is not special and does not show any imagination:

*The lyrics are boilerplate and uninspiring.*

-

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

**using System;**

**Console.WriteLine("Hello World!");**

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

**System.Console.WriteLine("Hello World!");**

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](https://docs.microsoft.com/en-us/azure/azure-functions/) is 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>.

For more information, see [Top-level statements](https://docs.microsoft.com/en-us/dotnet/csharp/fundamentals/program-structure/top-level-statements) in the C# Programming Guide.

**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:

public static bool IsLetter(this char c) =>

c is >= 'a' and <= 'z' or >= 'A' and <= 'Z';

With optional parentheses to make it clear that and has higher precedence than or:

public static bool IsLetterOrSeparator(this char c) =>

c is (>= 'a' and <= 'z') or (>= 'A' and <= 'Z') or '.' or ',';

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

if (e is not null)

{

// ...

}

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.

For more information, see [Patterns (C# reference)](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns).

For more information, see the [Relational patterns](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns#relational-patterns) and [Logical patterns](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns#logical-patterns) sections of the [Patterns](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/patterns) article.

**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](https://docs.microsoft.com/en-us/dotnet/api/system.intptr) and [System.UIntPtr](https://docs.microsoft.com/en-us/dotnet/api/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 run time. 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](https://docs.microsoft.com/en-us/dotnet/api/system.int32) and [System.UInt32](https://docs.microsoft.com/en-us/dotnet/api/system.uint32) types. If the result doesn't fit in 32 bits, the operation is executed at run time 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. For more information, see [nint and nuint types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/builtin-types/nint-nuint)

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. For more information, see [Unsafe code and pointer types](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/unsafe-code).

Finally, you can add the [System.Runtime.CompilerServices.SkipLocalsInitAttribute](https://docs.microsoft.com/en-us/dotnet/api/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](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.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. For more information, see [SkipLocalsInit attribute](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/general#skiplocalsinit-attribute).

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](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/new-operator) when the created object's type is already known. The most common use is in field declarations:

private List<WeatherObservation> \_observations = **new();**

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:

public WeatherForecast ForecastFor(DateTime forecastDate, WeatherForecastOptions options)

You could call it as follows:

var forecast = station.ForecastFor(DateTime.Now.AddDays(2), **new()**);

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

**WeatherStation station = new() { Location = "Seattle, WA" };**

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](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/conditional-operator). 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](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions) or [anonymous methods](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/delegate-operator). 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 [override](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/override) methods. An override method can return a type derived from the return type of the overridden base method. This can be useful for records and for other types that support virtual clone or factory methods.

In addition, the [foreach loop](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/statements/iteration-statements#the-foreach-statement) 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](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions#input-parameters-of-a-lambda-expression) section of the [Lambda expressions](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/operators/lambda-expressions) article.

Finally, you can now apply attributes to [local functions](https://docs.microsoft.com/en-us/dotnet/csharp/programming-guide/classes-and-structs/local-functions). For example, you can apply [nullable attribute annotations](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/nullable-analysis) 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. For more information, see [partial method (C# Reference)](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/keywords/partial-method).

The second new feature for code generators is ***module initializers***. Module initializers are methods that have the [ModuleInitializerAttribute](https://docs.microsoft.com/en-us/dotnet/api/system.runtime.compilerservices.moduleinitializerattribute) attribute attached to them. These methods will be called by the runtime before any other field access or method invocation within the entire module. 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. For more information, see [ModuleInitializer attribute](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/attributes/general#moduleinitializer-attribute).

-

-

-