# Introducing C# 9.0

C# 9.0 adds many new features, and focuses on a few themes. C# 9.0 is part of .NET 5, which continues the journey toward a single .NET ecosystem. The new features focus on modern workloads, that is, the software applications and services you’re building today. The C# 9.0 compiler ships with the .NET 5.0 SDK. Many of the C# 9.0 features rely on new features in the .NET 5.0 libraries and updates to the .NET CLR that’re part of .NET 5.0. Therefore, C# 9.0 is supported only on .NET 5.0. C# 9.0 focuses on features that support native cloud applications, modern software engineering practices, and more concise readable code. There are several new features that make up this release:

* Top-level statements
* Record types
* Init-only setters
* Enhancements to pattern matching
* Natural-sized integers
* Function pointers
* Omit localsinit
* Target type new
* Target type conditional
* Static anonymous methods
* Covariant return types
* Lambda discard parameters
* Attributes on local functions

This article explores those features and provides scenarios where you might use them.

## Top-Level Statements

Let’s start the exploration of C# 9.0 with top-level statements. This feature removes 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!");

}

}

}

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:

using System;

Console.WriteLine("Hello World!");

Top-level statements provide a way for you write programs with less ceremony. 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 line of code. None of the extra ceremony is needed. 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 provides. Top-level statements are great for small console programs and utilities. In addition, 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 or 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 example, the canonical Hello World example could be expanded to take an optional command line argument for a person’s name. If the argument is present, the program prints the name. If not, it prints “Hello World!” like this:

using System;

using System.Linq;

if (args.Any())

{

var msg =

args.Aggregate((s1, s2) => $"{s1} {s2}");

Console.WriteLine($"Hello {msg}");

}

else

Console.WriteLine("Hello World!");

## Record Types

Record types have a major impact on the code you write every day. There are a lot of smaller language enhancements that make up records. These enhancements are easier to understand by starting with the typical uses for records.

.NET types are largely classified as classes or anonymous types that are reference types and structs or tuples, which are value types. Although creating immutable value types is recommended, mutable value types don’t often introduce errors. That’s because value types are passed by value to methods, so any changes are made to a copy of the original data.

There are a lot of 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. For that reason, busy developers, which is all developers, write mutable 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 considers two records to be equal if their properties are all equal. Consider this definition:

public record Person

{

public string LastName { get; }

public string FirstName { get; }

public Person(string first, string last) =>

(FirstName, LastName) = (first, last);

}

That record definition creates a Person type that contains two read-only 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
* Deconstruct method

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

public record Teacher : Person

{

public string Subject { get; }

public Teacher(string first, string last,

string sub)

: base(first, last)

=> Subject = sub;

}

You can also seal records to prevent further derivation:

public sealed record Student : Person

{

public int Level { get; }

public Student(string first, string last,

int level)

: base(first, last)

=> Level = level;

}

The compiler generates different versions of the methods mentioned above depending on whether or not the record type is sealed and whether or not the direct base class is an object. The compiler does this to ensure that equality for records means that the record types match and the values of each property of the records are equal. Consider the small hierarchy above. The compiler generates methods so that a Person could not be considered equal to a Student or a Teacher. In addition to the familiar Equals overloads, operator == and operator !=, the compiler generates 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 the public properties declared in the base type and the record type. These synthesized methods enforce value-based equality throughout an inheritance hierarchy. That means that 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. If a record type is abstract, the clone method is also abstract. If a record type is sealed, the clone method is sealed. If the base type of the record is object, the clone method is virtual. Otherwise, it’s override. The result of all these rules is that you can create copies of a record ensuring that the copy is the same type. Furthermore, you can check the equality of any two records in an inheritance hierarchy and get the results that you intuitively expect.

Person p1 = new Person("Bill", "Wagner");

Student s1 = new Student("Bill", "Wagner", 11);

Console.WriteLine(s1 == p1); // false

The compiler synthesizes two methods that support printed output: a ToString override and PrintMembers. The PrintMembers method returns a comma-separated list of property names and values. The ToString() override returns the string produced by PrintMembers, surrounded by { and }. For example, the ToString method for Student generates a string like the following:

Student { LastName = Wagner, FirstName = Bill,

Level = 11 }

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 using positional record syntax:

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, string Subject)

: Person(FirstName, LastName);

These declarations create the same functionality as the earlier version (with a couple extra features that I’ll cover in a bit). 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:

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.

// Deconstruct is in the order

// declared in the record.

(string first, string last) = p1;

Console.WriteLine(first);

Console.WriteLine(last);

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

Person brother = p1 with { FirstName = "Paul" };

The above line creates a new Person record where the LastName property is a copy of p1, and the first Name 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 generate 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. The main benefit is that callers can use property initializer syntax to set these values in creation expressions, but those properties are read-only 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, has completed. Consider this immutable Point structure:

public struct Point

{

public double X { get; init; }

public double Y { get; init; }

public double Distance => Math.Sqrt(X \* X +

Y \* Y);

}

You can initialize this structure using property initializer syntax, but you can’t modify the values once construction and initialization has completed:

var pt = new Point { X = 3, Y = 4 };

pt.X = 7; // Error!

Console.WriteLine(pt.Distance);

In addition to using property initializers, init-only setters can be very useful to set base class properties from derived classes, or 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.

## Enhanced Pattern Matching

C# 9 includes new pattern-matching improvements:

* Type patterns match a variable as 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 that 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';

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

public static bool IsLetterIsSeparator(this char c) =>

c is (>= 'a' and <= 'z')

or (>= 'A' and <= 'Z')

or '.'

or ',';

One of the most common uses is a new clear 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.

## 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 are 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 ints don’t have constants for MaxValue or MinValue, except for nuint.MinValue, which has a MinValue of 0. Other values cannot be expressed as constants because it would depend on the native size of an integer on the target computer. 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 Int32 and 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 faster 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 SkipLocalsInitAttribute to instruct the compiler not to emit the localsinit flag. This flag instructs the CLR to zero-initialize all local variables. This 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 does not 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:

public class PropertyBag

{

private Dictionary<string, object>

properties = new();

// elided

}

Target type new can also be used when you need to create a new object to pass as a parameter to a method. Consider a ParseJson() method with the following signature:

public JsonElement ParseJson(string text,

JsonSerializerOptions opts)

You could call it as follows:

var result = ParseJson(text, new());

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 common 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. That has the same effect as the “static” modifier on local functions: a static lambda or anonymous function can’t capture local variables or instance state. This prevents accidentally capturing other variables.

Covariant return types provide flexibility for the return types of virtual functions. Previously, overrides had to return the same type as the base function. Now, overrides may return a type derived from the return type of the base function. This can be useful for Records and for other types that support virtual clone or factory methods.

Next, you can use discards as parameters to lambda expressions. This convenience enables you to avoid naming the argument, and the compiler may be able to avoid using it. You use the “\_” for any argument.

Finally, you can now apply attributes to local functions. These are particularly useful to add nullable attribute annotations to local functions.

## Support for Source Generators

Two final features support C# source generators. C# source generators are a component you can write that’s similar to a Roslyn analyzer or code fix. The difference is that source generators analyze code and write new source code files as part of the compilation process. A typical source generator searches code for attributes or other conventions. Based on the information supplied by the attributes, the source generator writes new code that’s added to the library or application.

You can read attributes or other code elements using the Roslyn analysis APIs. From that information, you can add new code to the compilation. Source generators can only add code; they’re not allowed to modify any existing code in the compilation.

The two features added for source generators are extensions to partial method syntax and module initializers. First, there are fewer restrictions to partial methods. Before C# 9.0, partial methods were private but can’t specify an access modifier, have a void return, or have out parameters. These restrictions meant that if no method implementation was provided, the compiler removed all calls to the partial method. C# 9.0 removes these restrictions but requires that partial method declarations have an implementation. Source 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 source generators is module initializers. These are methods that have the System.Runtime.CompilerServices.ModuleInitializer attribute attached to them. These methods are 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 cannot be a local function. Source generators may need to generate initialization code. Module initializers provide a standard place for that code to reside.

## Summary

C# 9.0 continues the evolution of C# as a modern language. It’s embracing new idioms and new programming paradigms while retaining its roots as an object-oriented component-based language. The new features make it efficient to build the modern programs that we’re creating today. Try to adopt the new features today.

Bill Wagner

[wiwagn@microsoft.com](mailto:wiwagn@microsoft.com)   
https://twitter.com/billwagner

Bill Wagner is responsible for the C# area of [https://docs.microsoft.com](https://docs.microsoft.com/). He creates learning materials for developers interested in the C# language and .NET. He's a member of the ECMA C# Standards Committee and the .NET Foundation board of directors. He is President of the Humanitarian Toolbox. Before joining Microsoft, he was awarded Microsoft Regional Director and .NET MVP for 10+years. He is the author of *Effective C#* and *More Effective C#*.