# 5 Features in C# 10 Every Developer Should Know

* [Selva Kumar Ramarasu](https://www.syncfusion.com/blogs/author/selvakumar-ramarasu)
* October 3, 2022**UPDATED**

C# is one of the oldest and fast-growing programming languages in the world. This blog lists the top five features in C# 10 that help developers write code faster and prettier:

1. [Constant interpolated strings](https://www.syncfusion.com/blogs/post/5-features-in-c-sharp-10-every-developer-should-know.aspx#constant-interpolated-strings).
2. [Extended property patterns](https://www.syncfusion.com/blogs/post/5-features-in-c-sharp-10-every-developer-should-know.aspx#extended-property-patterns).
3. [Global using](https://www.syncfusion.com/blogs/post/5-features-in-c-sharp-10-every-developer-should-know.aspx#global-using).
4. [File-scoped namespace](https://www.syncfusion.com/blogs/post/5-features-in-c-sharp-10-every-developer-should-know.aspx#file-scoped-namespace).
5. [Assignment and declaration in the same deconstruction](https://www.syncfusion.com/blogs/post/5-features-in-c-sharp-10-every-developer-should-know.aspx#assignment-and-declaration-in-the-same-deconstruction).

Let’s look at each of these features.

**Note:** With the details provided in the [C# language versioning](https://docs.microsoft.com/en-us/dotnet/csharp/language-reference/configure-language-version), C# 10 is supported only on .NET 6 and newer versions.

## Constant interpolated strings

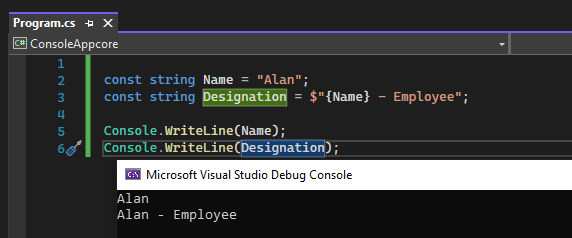
String interpolation, introduced in C# 6, provides an easy and convenient way to apply syntax to a string. An interpolated string is usually a combination of strings and an expression. On resolving an interpolated string to a string, the expression is executed, and the resultant string value is added in its place.

Back then, string interpolation was restricted only to strings, and a string declared as a constant cannot be interpolated. But now in C# 10, this string interpolation feature is extended to strings declared as constants as well, with the condition that only constant strings can be used in the expression.

Look at the following code example for more clarity.

const string Name = "Alan";

const string Designation = $"{Name} - Employee";



## Extended property patterns

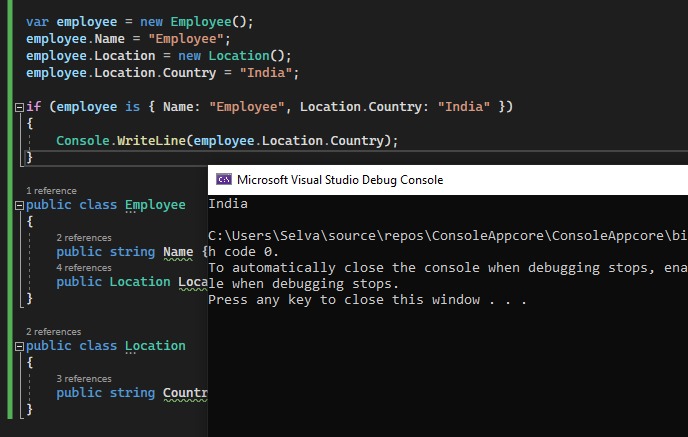
Extended property patterns help us to improve the readability of code in accessing a child property from a parent property. In C# versions prior to 10, the child properties were inaccessible at the same level.

### C# 8

{ ParentProperty: { ChildProperty: Value } }

### C# 10

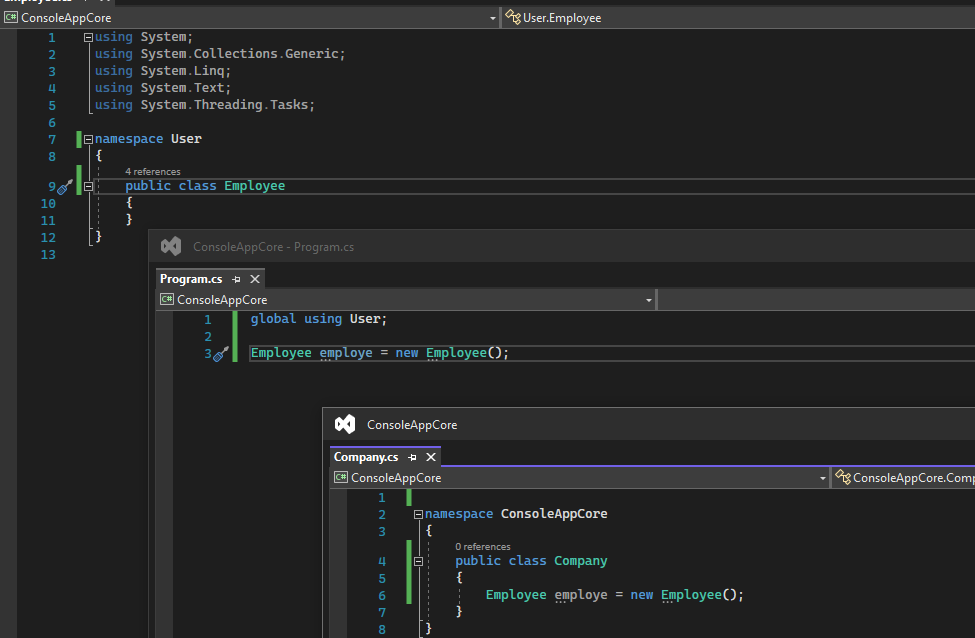
{ ParentProperty.ChildProperty: Value }

Extended Property Patterns in C# 10

## Global using

Use the new global using feature in the C# 10, to avoid referring to the same directive in each class and page. Adding the **global** modifier as a prefix to the **using** namespace or directive will apply the reference to all files in the project.

global using System;

Global Using in C# 10

## File-scoped namespace

With the new file-scoped namespace declaration in C# 10, you can declare a single namespace for the entire file. After declaring the file-scoped namespace, we can’t declare any other namespace. You can only declare and define the following types:

* Class
* Struct
* Delegate
* Enum
* Interface

### Old format

namespace ConsoleAppcore

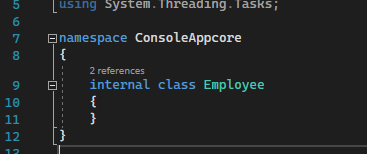
{

internal class User

{

}

}



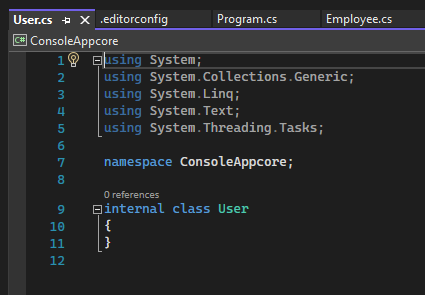
### New format for C# 10

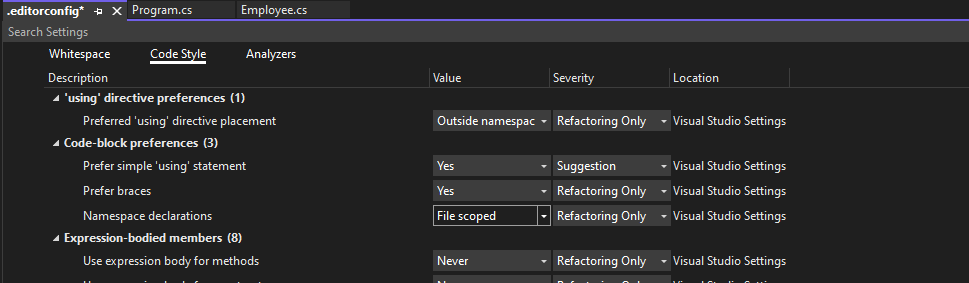
namespace ConsoleAppcore;

internal class User

{

}

  
To use this feature in all the new classes to be created, follow these steps in Visual Studio 2022:

1. First, right-click the project. Then, choose **Add** > **New EditorConfig.**
2. Now, open the Editor configuration file.
3. Move to the **Code Style** tab.
4. Finally, change **Namespace declarations** to **File Scoped** from **Block Spaced** and save the file.
5. Hereafter, creating a new CS file will use the **file-scoped** format.

## Assignment and declaration in the same deconstruction

In the previous versions of C#, developers can either assign values to existing variables or declare new variables in deconstruction. The combination of assigning a value to an existing variable and declaring a new variable was restricted. This restriction is now removed, and you can assign a value and declare a new variable in a single deconstruction.

### Before C# 10

(string name, string email) = var employee;

Or

string name;

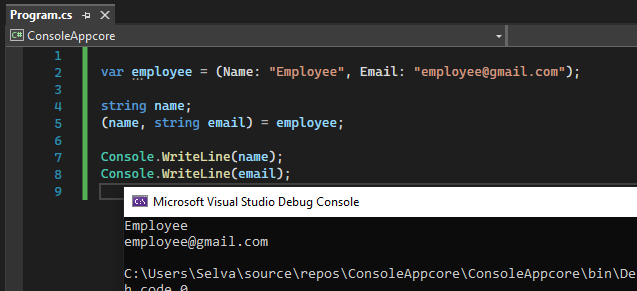
string email;

(name, email) = employee;

### In C# 10

string name;

(name, string email) = employee;



# Welcome to C# 10

Today, we are happy to announce the release of C# 10 as part of .NET 6 and Visual Studio 2022. In this post, we’re covering a lot of the new C# 10 features that make your code prettier, more expressive, and faster.

Read the [Visual Studio 2022 announcement](https://aka.ms/vs2022gablog) and the [.NET 6 announcement](https://aka.ms/dotnet6-GA) to find out more, including how to install.

## Global and implicit usings

using directives simplify how you work with namespaces. C# 10 includes a new global using directive and implicit usings to reduce the number of usings you need to specify at the top of each file.

### Global using directives

If the keyword global appears prior to a using directive, that using applies to the entire project:

global using System;

You can use any feature of using within a global using directive. For example, adding static imports a type and makes the type’s members and nested types available throughout your project. If you use an alias in your using directive, that alias will also affect your entire project:

global using static System.Console;

global using Env = System.Environment;

You can put global usings in any .cs file, including Program.cs or a specifically named file like globalusings.cs. The scope of global usings is the current compilation, which generally corresponds to the current project.

For more information, see [global using directives](https://docs.microsoft.com/dotnet/csharp/language-reference/keywords/using-directive#global-modifier).

### Implicit usings

The Implicit usings feature automatically adds common global using directives for the type of project you are building. To enable implicit usings set the ImplicitUsings property in your .csproj file:

<PropertyGroup>

<!-- Other properties like OutputType and TargetFramework -->

<ImplicitUsings>enable</ImplicitUsings>

</PropertyGroup>

Implicit usings are enabled in the new .NET 6 templates. Read more about the changes to the .NET 6 templates at this [blog post](https://devblogs.microsoft.com/dotnet/announcing-net-6-preview-7/#net-sdk-c-project-templates-modernized).

The specific set of global using directives included depend on the type of application you are building. For example, implicit usings for a console application or a class library are different than those for an ASP.NET application.

For more information, see this [implicit usings](https://aka.ms/csharp-implicit-usings) article.

### Combining using features

Traditional using directives at the top of your files, global using directives, and implicit usings work well together. Implicit usings let you include the .NET namespaces appropriate to the kind of project you’re building with a single line in your project file. global using directives let you include additional namespaces to make them available throughout your project. The using directives at the top of your code files let you include namespaces used by just a few files in your project.

Regardless of how they are defined, extra using directives increase the possibility of ambiguity in name resolution. If you encounter this, consider adding an alias or reducing the number of namespaces you are importing. For example, you can replace global using directives with explicit using directives at the top of a subset of files.

If you need to remove namespaces that have been included via implicit usings, you can specify them in your project file:

<ItemGroup>

<Using Remove="System.Threading.Tasks" />

</ItemGroup>

You can also add namespace that behave as though they were global using directives, you can add Using items to your project file, for example:

<ItemGroup>

<Using Include="System.IO.Pipes" />

</ItemGroup>

## File-scoped namespaces

Many files contain code for a single namespace. Starting in C# 10, you can include a namespace as a statement, followed by a semi-colon and without the curly brackets:

namespace MyCompany.MyNamespace;

class MyClass // Note: no indentation

{ ... }

This simplifies the code and removes a level of nesting. Only one file-scoped namespace declaration is allowed, and it must come before any types are declared.

For more information about file-scoped namespaces, see the [namespace keyword](https://docs.microsoft.com/dotnet/csharp/language-reference/keywords/namespace) article.

## Improvements for lambda expressions and method groups

We’ve made several improvements to both the types and the [syntax surrounding lambdas](https://docs.microsoft.com/dotnet/csharp/whats-new/csharp-10#lambda-expression-improvements). We expect these to be widely useful, and one of the driving scenarios has been to make [ASP.NET Minimal APIs](https://devblogs.microsoft.com/dotnet/announcing-asp-net-core-in-net-6/) even more straightforward.

### Natural types for lambdas

Lambda expressions now sometimes have a “natural” type. This means that the compiler can often infer the type of the lambda expression.

Up until now a lambda expression had to be converted to a delegate or an expression type. For most purposes you’d use one of the overloaded Func<...> or Action<...> delegate types in the BCL:

Func<string, int> parse = (string s) => int.Parse(s);

Starting with C# 10, however, if a lambda does not have such a “target type” we will try to compute one for you:

var parse = (string s) => int.Parse(s);

You can hover over var parse in your favorite editor and see that the type is still Func<string, int>. In general, the compiler will use an available Func or Action delegate, if a suitable one exists. Otherwise, it will synthesize a delegate type (for example, when you have ref parameters or have a large number of parameters).

Not all lambdas have natural types – some just don’t have enough type information. For instance, leaving off parameter types will leave the compiler unable to decide which delegate type to use:

var parse = s => int.Parse(s); // ERROR: Not enough type info in the lambda

The natural type of lambdas means that they can be assigned to a weaker type, such as object or Delegate:

object parse = (string s) => int.Parse(s); // Func<string, int>

Delegate parse = (string s) => int.Parse(s); // Func<string, int>

When it comes to expression trees we do a combination of “target” and “natural” typing. If the target type is LambdaExpression or the non-generic Expression (base type for all expression trees) and the lambda has a natural delegate type D we will instead produce an Expression<D>:

LambdaExpression parseExpr = (string s) => int.Parse(s); // Expression<Func<string, int>>

Expression parseExpr = (string s) => int.Parse(s); // Expression<Func<string, int>>

### Natural types for method groups

Method groups (that is, method names without argument lists) now also sometimes have a natural type. You have always been able to convert a method group to a compatible delegate type:

Func<int> read = Console.Read;

Action<string> write = Console.Write;

Now, if the method group has just one overload it will have a natural type:

var read = Console.Read; // Just one overload; Func<int> inferred

var write = Console.Write; // ERROR: Multiple overloads, can't choose

### Return types for lambdas

In the previous examples, the return type of the lambda expression was obvious and was just being inferred. That isn’t always the case:

var choose = (bool b) => b ? 1 : "two"; // ERROR: Can't infer return type

In C# 10, you can specify an explicit return type on a lambda expression, just like you do on a method or a local function. The return type goes right before the parameters. When you specify an explicit return type, the parameters must be parenthesized, so that it’s not too confusing to the compiler or other developers:

var choose = object (bool b) => b ? 1 : "two"; // Func<bool, object>

### Attributes on lambdas

Starting in C# 10, you can put attributes on lambda expressions in the same way you do for methods and local functions. They go right where you expect; at the beginning. Once again, the lambda’s parameter list must be parenthesized when there are attributes:

Func<string, int> parse = [Example(1)] (s) => int.Parse(s);

var choose = [Example(2)][Example(3)] object (bool b) => b ? 1 : "two";

Just like local functions, attributes can be applied to lambdas if they are valid on AttributeTargets.Method.

Lambdas are invoked differently than methods and local functions, and as a result attributes do not have any effect when the lambda is invoked. However, attributes on lambdas are still useful for code analysis, and they are also emitted on the methods that the compiler generates under the hood for lambdas, so they can be discovered via reflection.

## Improvements to structs

C# 10 introduces features for structs that provide better parity between structs and classes. These new features include parameterless constructors, field initializers, record structs and with expressions.

### Parameterless struct constructors and field initializers

Prior to C# 10, every struct had an implicit public parameterless constructor that set the struct’s fields to default. It was an error for you to create a parameterless constructor on a struct.

Starting in C# 10, you can include your own parameterless struct constructors. If you don’t supply one, the implicit parameterless constructor will be supplied to set all fields to their default. Parameterless constructors you create in structs must be public and cannot be partial:

public struct Address

{

public Address()

{

City = "<unknown>";

}

public string City { get; init; }

}

You can initialize fields in a parameterless constructor as above, or you can initialize them via field or property initializers:

public struct Address

{

public string City { get; init; } = "<unknown>";

}

Structs that are created via default or as part of array allocation ignore explicit parameterless constructors, and always set struct members to their default values. For more information about parameterless constructors in structs, see the [struct type](https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/struct#parameterless-constructors-and-field-initializers).

### record structs

Starting in C# 10, records can now be defined with record struct. These are similar to record classes that were introduced in C# 9:

public record struct Person

{

public string FirstName { get; init; }

public string LastName { get; init; }

}

You can continue to define record classes with record, or you can use record class for clarity.

Structs already had value equality – when you compare them it is by value. Record structs add IEquatable<T> support and the == operator. Record structs provide a custom implementation of IEquatable<T> to avoid the performance issues of reflection, and they include record features like a ToString() override.

Record structs can be positional, with a primary constructor implicitly declaring public members:

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

The parameters of the primary constructor become public auto-implemented properties of the record struct. Unlike record classes, the implicitly created properties are read/write. This makes it easier to convert tuples to named types. Changing return types from a tuple like (string FirstName, string LastName) to a named type of Person can clean up your code and guarantee consistent member names. Declaring the positional record struct is easy and keeps the mutable semantics.

If you declare a property or field with the same name as a primary constructor parameter, no auto-property will be synthesized and yours will be used.

To create an immutable record struct, add readonly to the struct (as you can to any struct) or apply readonly to individual properties. Object initializers are part of the construction phase where readonly properties can be set. Here is just one of the ways you can work with immutable record structs:

var person = new Person { FirstName = "Mads", LastName = "Torgersen"};

public readonly record struct Person

{

public string FirstName { get; init; }

public string LastName { get; init; }

}

Find out more about [record structs in this article](https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/record).

### sealed modifier on ToString() in record classes

Record classes have also been improved. Starting in C# 10 the ToString() method can include the sealed modifier, which prevents the compiler from synthesizing a ToString implementation for any derived records.

Find out more about ToString() in records [in this article](https://docs.microsoft.com/dotnet/csharp/language-reference/builtin-types/record#built-in-formatting-for-display).

### with expressions on structs and anonymous types

C# 10 supports with expressions for all structs, including record structs, as well as for anonymous types:

var person2 = person with { LastName = "Kristensen" };

This returns a new instance with the new value. You can update any number of values. Values you do not set will retain the same value as the initial instance.

Learn more about with [in this article](https://docs.microsoft.com/dotnet/csharp/language-reference/operators/with-expression)

## Interpolated string improvements

When we added interpolated strings to C#, we always felt that there was more that could be done with that syntax down the line, both for performance and expressiveness. With C# 10, that time has come!

### Interpolated string handlers

Today the compiler turns interpolated strings into a call to string.Format. This can lead to a lot of allocations – the boxing of arguments, allocation of an argument array, and of course the resulting string itself. Also, it leaves no wiggle room in the meaning of the actual interpolation.

In C# 10 we’ve added a library pattern that allows an API to “take over” the handling of an interpolated string argument expression. As an example, consider StringBuilder.Append:

var sb = new StringBuilder();

sb.Append($"Hello {args[0]}, how are you?");

Up until now, this would call the Append(string? value) overload with a newly allocated and computed string, appending that to the StringBuilder in one chunk. However, Append now has a new overload Append(ref StringBuilder.AppendInterpolatedStringHandler handler) which takes precedence over the string overload when an interpolated string is used as argument.

In general, when you see parameter types of the form SomethingInterpolatedStringHandler the API author has done some work behind the scenes to handle interpolated strings more appropriately for their purposes. In the case of our Append example, the strings "Hello ", args[0] and ", how are you?" will be individually appended to the StringBuilder, which is much more efficient and has the same outcome.

Sometimes you want to do the work of building the string only under certain conditions. An example is Debug.Assert:

Debug.Assert(condition, $"{SomethingExpensiveHappensHere()}");

In most cases, the condition will be true and the second parameter is unused. However, all of the arguments are computed on every call, needlessly slowing down execution. Debug.Assert now has an overload with a custom interpolated string builder, which ensures that the second argument isn’t even evaluated unless the condition is false.

Finally, here’s an example of actually changing the behavior of string interpolation in a given call: String.Create() lets you specify the IFormatProvider used to format the expressions in the holes of the interpolated string argument itself:

String.Create(CultureInfo.InvariantCulture, $"The result is {result}");

You can learn more about interpolated string handlers, [in this article](https://docs.microsoft.com/dotnet/csharp/language-reference/tokens/interpolated#compilation-of-interpolated-strings) and this [tutorial](https://docs.microsoft.com/dotnet/csharp/whats-new/tutorials/interpolated-string-handler) on creating a custom handler.

### Constant interpolated strings

If all the holes of an interpolated string are constant strings, then the resulting string is now also constant. This lets you use string interpolation syntax in more places, like attributes:

[Obsolete($"Call {nameof(Discard)} instead")]

Note that the holes must be filled with constant strings. Other types, like numeric or date values, cannot be used because they are sensitive to Culture, and can’t be computed at compile time.

## Other improvements

C# 10 has a number of smaller improvements across the language. Some of these just make C# work in the way you expect.

### Mix declarations and variables in deconstruction

Prior to C# 10, deconstruction required all variables to be new, or all of them to be previously declared. In C# 10, you can mix:

int x2;

int y2;

(x2, y2) = (0, 1); // Works in C# 9

(var x, var y) = (0, 1); // Works in C# 9

(x2, var y3) = (0, 1); // Works in C# 10 onwards

Find out more in the article on [deconstruction](https://docs.microsoft.com/dotnet/csharp/fundamentals/functional/deconstruct).

### Improved definite assignment

C# produces errors if you use a value that has not been definitely assigned. C# 10 understands your code better and produces less spurious errors. These same improvements also mean you’ll see less spurious errors and warnings for null references.

Find out more about C# definite assignment in the [what’s new in C# 10 article](https://docs.microsoft.com/dotnet/csharp/whats-new/csharp-10#improved-definite-assignment).

### Extended property patterns

C# 10 adds extended property patterns to make it easier to access nested property values in patterns. For example, if we add an address to the Person record above, we can pattern match in both of the ways shown here:

object obj = new Person

{

FirstName = "Kathleen",

LastName = "Dollard",

Address = new Address { City = "Seattle" }

};

if (obj is Person { Address: { City: "Seattle" } })

Console.WriteLine("Seattle");

if (obj is Person { Address.City: "Seattle" }) // Extended property pattern

Console.WriteLine("Seattle");

The extended property pattern simplifies the code and makes it easier to read, particularly when matching against multiple properties.

Find out more about extended property patterns in the [pattern matching article](https://docs.microsoft.com/dotnet/csharp/language-reference/operators/patterns#property-pattern).

### Caller expression attribute

CallerArgumentExpressionAttribute supplies information about the context of a method call. Like the other CompilerServices attributes, this attribute is applied to an optional parameter. In this case, a string:

void CheckExpression(bool condition,

[CallerArgumentExpression("condition")] string? message = null )

{

Console.WriteLine($"Condition: {message}");

}

The parameter name passed to CallerArgumentExpression is the name of a different parameter. The expression passed as the argument to that parameter will be contained in the string. For example,

var a = 6;

var b = true;

CheckExpression(true);

CheckExpression(b);

CheckExpression(a > 5);

C# 10 Features

I’m going to talk about every single C# 10 features. We will look at them individually with an example.

The top 6 features are listed down.

1. Global using directive
2. File Scoped Namespace
3. Constant Interpolated strings
4. Lambda Improvements
5. Null Parameter Checking

IDE which I have used for preparing the examples

* Microsoft Visual Studio Community 2022 (64-bit)
* Version:  17.0.5

Global using Directive

This is one of the interesting features. Let’s go ahead and create a Console application in VS 2022 and add the below lines into Program.cs file

using System.Text.Json;

var names = new[] { "Prasad", "Praveen" };

var serialized = JsonSerializer.Serialize(names);

Console.WriteLine(serialized);

C#

Copy

This code is perfectly valid in C# 9.0.

Assume that we need to utilize the JsonSerializer.Serialize() method in multiple files under the same project, Till C# 9.0, we have only one option to add the reference System,Text.Json at the beginning of each files. In C# 10, Microsoft has been introduced a new feature to add the reference globally at one place using the keyword “global”. It is recommended to create a separate file which will contain these imports, something like “usings.cs”

C# 10 code would look like below

**Program.cs**

var names = new[] { "Prasad", "Praveen" };

var serialized = JsonSerializer.Serialize(names);

Console.WriteLine(serialized);

C#

Copy

A new file usings.cs will be created and moved all the using imports into it.

**Usings.cs**

This class file will have only global **using System.Text.Json**

Assume that I’m going to comment out the line in Usings.cs, if you go back to Program.cs class, you will be getting a compiler error. To resolve this issue, we can edit the project file, and add the **<ItemGroup>** tags

GlobalUsingDirective.csproj( this is my project file)

<PropertyGroup>

<OutputType>Exe</OutputType>

<TargetFramework>net6.0</TargetFramework>

<ImplicitUsings>enable</ImplicitUsings>

<Nullable>enable</Nullable>

</PropertyGroup>

<ItemGroup Condition="'$(ImplicitUsings)'=='true' Or '$(ImplicitUsings)'== 'enable'">

<Using Include="System.Text.Json"/>

</ItemGroup>

</Project>

Markup

Copy

File Scoped Namespaces

Assume that I have a class file Employee.cs and inside the class, the code looks like below

namespace FileScopedNamespace

{

internal class Employee

{

public string Name { get; set; }

}

}

C#

Copy

I’m going to add a new namespace within the same file as below, now my Employee.cs file looks like below since in theory, we can have multiple namespaces in a file.

namespace FileScopedNamespace

{

internal class Employee

{

public string Name { get; set; }

}

}

namespace FileScopedNamespace1

{

internal class Employee1

{

public string Name { get; set; }

}

}

C#

Copy

In my 10 years of career in C#, I have never seen such a class file with multiple namespaces. I assume, C# Dev Team also may have same thoughts and what they did, they just removed the pointless nesting of namespace and class and converted into File Scoped Namespace as below

namespace FileScopedNamespace;

internal class Employee

{

public string Name { get; set; }

}

C#

Copy

Whatever we are in this Employee.cs file comes under the namespace "FileScopedNamespace;".   This is really amazing, and I love this feature.

Constant Interpolated strings

This is another feature which I really like. Assume I have a class like below, I’m trying to apply string interpolation on constant variables.

namespace ConstantInterpolatedString

{

public static class EmailMessages

{

private const string Salutation = "Welcome";

public static class Header

{

public const string SalutionTeamplate = Salutation + " to constant Interpolation";

}

}

}

C#

Copy

This can be converted into Constant interpolation as below

namespace ConstantInterpolatedString

{

public static class EmailMessages

{

private const string Salutation = "Welcome";

public static class Header

{

public const string SalutionTeamplate = $"{Salutation} to Interpolation";

}

}

}

C#

Copy

Before C# 10, there was no option to apply interpolation on const variables. It will throw the below compiler error

The expression being assigned to EmailMessages.Header.SalutionTemplate must be constant.

Lambda Improvements

Here we will look at some of the lambda improvements.

Previously, we had a function as below

using System;

Func<string> welcome = () => "Welcome to Lambda Improvements";

Console.WriteLine(welcome);

C#

Copy

Here we have a func delegate which returns a string.

Note:

Ensure that our language version is pointing to 9.0 in project file as below

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<OutputType>Exe</OutputType>

<TargetFramework>net6.0</TargetFramework>

<ImplicitUsings>enable</ImplicitUsings>

<Nullable>enable</Nullable>

<LangVersion>9.0</LangVersion>

</PropertyGroup>

</Project>

Markup

Copy

Now let’s change the syntax of func delegate as below

using System;

var welcome = () => "Welcome to Lambda Improvements";

Console.WriteLine(welcome);

C#

Copy

You will get an error as below

***Feature ‘inferred delegate type’ is not available in C# 9.0. Please use language version 10.0 or greater.***

Now, update the language version in project file and see.

<Project Sdk="Microsoft.NET.Sdk">

<PropertyGroup>

<OutputType>Exe</OutputType>

<TargetFramework>net6.0</TargetFramework>

<ImplicitUsings>enable</ImplicitUsings>

<Nullable>enable</Nullable>

<LangVersion>10.0</LangVersion>

</PropertyGroup>

</Project>

Markup

Copy

The code will work as expected.

Next improvement,

Assume that we have the below code

using System;

//var welcome = () => "Welcome to Lambda Improvements";

var text = () => null;

//Console.WriteLine(welcome);

C#

Copy

This will raise exception as below

***The delegate type could not be inferred***

There is an alternative as below

using System;

//var welcome = () => "Welcome to Lambda Improvements";

Func<string?> test = () => null;

//Console.WriteLine(welcome);

C#

Copy

This works fine. However, I want to rewrite this line using var. how? Please see the below syntax

using System;

//var welcome = () => "Welcome to Lambda Improvements";

var text = string? () => null;

//Console.WriteLine(welcome);

C#

Copy

This code will build with zero errors.

Null Parameter Checking

In C # 9.0, we can write the null check as below

string? text = null;

SayHello(text);

void SayHello(string message)

{

if(message is null)

{

throw new ArgumentNullException(nameof(message));

}

Console.WriteLine(message);

}

C#

Copy

In C# 10.0, we can replace the if condition with below one-liner code

string? text = null;

SayHello(text);

void SayHello(string message)

{

//if(message is null)

//{

// throw new ArgumentNullException(nameof(message));

//}

ArgumentNullException.ThrowIfNull(message);

Console.WriteLine(message);

}

C#

Copy

Thank you for reading my article. I will come up with other features in the upcoming articles. Please provide your feedback in the comment box.