Level Up Sessie C# 10

[What's new in C# 10 - C# Guide | Microsoft Docs](https://docs.microsoft.com/nl-nl/dotnet/csharp/whats-new/csharp-10)(, 2021)

[Explore string interpolation handlers | Microsoft Docs](https://docs.microsoft.com/nl-nl/dotnet/csharp/whats-new/tutorials/interpolated-string-handler)(, 2021)

[Lambda expressions - C# reference | Microsoft Docs](https://docs.microsoft.com/nl-nl/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression)(, 2022)

[const keyword - C# Reference | Microsoft Docs](https://docs.microsoft.com/nl-nl/dotnet/csharp/language-reference/keywords/const)(, 2021)

## Assignment and declaration in same deconstruction

This change removes a restriction from earlier versions of C#. Previously, a deconstruction could assign all values to existing variables, or initialize newly declared variables:

C#Kopiëren

// Initialization:

(int x, int y) = point;

// assignment:

int x1 = 0;

int y1 = 0;

(x1, y1) = point;

C# 10 removes this restriction:

C#Kopiëren

int x = 0;

(x, int y) = point;

## Improved definite assignment

Prior to C# 10, there were many scenarios where definite assignment and null-state analysis produced warnings that were false positives. These generally involved comparisons to boolean constants, accessing a variable only in the true or false statements in an if statement, and null coalescing expressions. These examples generated warnings in previous versions of C#, but don't in C# 10:

C#Kopiëren

string representation = "N/A";

if ((c != null && c.GetDependentValue(out object obj)) == true)

{

representation = obj.ToString(); // undesired error

}

// Or, using ?.

if (c?.GetDependentValue(out object obj) == true)

{

representation = obj.ToString(); // undesired error

}

// Or, using ??

if (c?.GetDependentValue(out object obj) ?? false)

{

representation = obj.ToString(); // undesired error

}

The main impact of this improvement is that the warnings for definite assignment and null-state analysis are more accurate.

## Generic attributes

**Belangrijk**

Generic attributes is a preview feature. You must [**set <LangVersion> to Preview**](https://docs.microsoft.com/nl-nl/dotnet/csharp/language-reference/compiler-options/language#langversion) to enable this feature. This feature may change before its final release.

You can declare a [generic class](https://docs.microsoft.com/nl-nl/dotnet/csharp/programming-guide/generics/generic-classes) whose base class is [System.Attribute](https://docs.microsoft.com/en-us/dotnet/api/system.attribute). This provides a more convenient syntax for attributes that require a [System.Type](https://docs.microsoft.com/en-us/dotnet/api/system.type) parameter. Previously, you'd need to create an attribute that takes a Type as its constructor parameter:

C#Kopiëren

public class TypeAttribute : Attribute

{

public TypeAttribute(Type t) => ParamType = t;

public Type ParamType { get; }

}

And to apply the attribute, you use the [typeof](https://docs.microsoft.com/nl-nl/dotnet/csharp/language-reference/operators/type-testing-and-cast#typeof-operator) operator:

C#Kopiëren

[TypeAttribute(typeof(string))]

public string Method() => default;

Using this new feature, you can create a generic attribute instead:

C#Kopiëren

public class GenericAttribute<T> : Attribute { }

Then, specify the type parameter to use the attribute:

C#Kopiëren

[GenericAttribute<string>()]

public string Method() => default;

You can apply a fully closed constructed generic attribute. In other words, all type parameters must be specified. For example, the following is not allowed:

C#Kopiëren

public class GenericType<T>

{

[GenericAttribute<T>()] // Not allowed! generic attributes must be fully closed types.

public string Method() => default;

}

The type arguments must satisfy the same restrictions as the [typeof](https://docs.microsoft.com/nl-nl/dotnet/csharp/language-reference/operators/type-testing-and-cast#typeof-operator) operator. Types that require metadata annotations aren't allowed. Examples include the following:

* dynamic
* nint, nuint
* string? (or any nullable reference type)
* (int X, int Y) (or any other tuple types using C# tuple syntax).

These types aren't directly represented in metadata. They include annotations that describe the type. In all cases, you can use the underlying type instead:

* object for dynamic.
* [IntPtr](https://docs.microsoft.com/en-us/dotnet/api/system.intptr) instead of nint or unint.
* string instead of string?.
* ValueTuple<int, int> instead of (int X, int Y).

Verwijzingen

(2021). *const (C# Reference).*  microsoft.com. https://docs.microsoft.com/nl-nl/dotnet/csharp/language-reference/keywords/const

(2021). *Tutorial: Write a custom string interpolation handler.*  microsoft.com. https://docs.microsoft.com/nl-nl/dotnet/csharp/whats-new/tutorials/interpolated-string-handler

(2022). *Lambda expressions (C# reference).*  microsoft.com. https://docs.microsoft.com/nl-nl/dotnet/csharp/language-reference/operators/lambda-expressions#natural-type-of-a-lambda-expression

(2021). *What's new in C# 10.*  microsoft.com. https://docs.microsoft.com/nl-nl/dotnet/csharp/whats-new/csharp-10