Miscellaneous attributes interpreted by the C# compiler

Article • 03/15/2023

The attributes `Conditional`, `Obsolete`, `AttributeUsage`, `AsyncMethodBuilder`, `InterpolatedStringHandler`, and `ModuleInitializer` can be applied to elements in your code. They add semantic meaning to those elements. The compiler uses those semantic meanings to alter its output and report possible mistakes by developers using your code.

`Conditional` attribute

The `Conditional` attribute makes the execution of a method dependent on a preprocessing identifier. The `Conditional` attribute is an alias for ConditionalAttribute, and can be applied to a method or an attribute class.

In the following example, `Conditional` is applied to a method to enable or disable the display of program-specific diagnostic information:

C#

```csharp
#define TRACE_ON
using System.Diagnostics;

namespace AttributeExamples;

public class Trace
{
    [Conditional("TRACE_ON")]
    public static void Msg(string msg)
    {
        Console.WriteLine(msg);
    }
}

public class TraceExample
{
    public static void Main()
    {
        Trace.Msg("Now in Main...");
        Console.WriteLine("Done.");
    }
}
```

If the `TRACE_ON` identifier isn't defined, the trace output isn't displayed. Explore for yourself in the interactive window.

The `Conditional` attribute is often used with the `DEBUG` identifier to enable trace and logging features for debug builds but not in release builds, as shown in the following example:

C#

```
[Conditional("DEBUG")]
static void DebugMethod()
{
}
```

When a method marked conditional is called, the presence or absence of the specified preprocessing symbol determines whether the compiler includes or omits calls to the method. If the symbol is defined, the call is included; otherwise, the call is omitted. A conditional method must be a method in a class or struct declaration and must have a `void` return type. Using `Conditional` is cleaner, more elegant, and less error-prone than enclosing methods inside `#if…#endif` blocks.

If a method has multiple `Conditional` attributes, compiler includes calls to the method if one or more conditional symbols are defined (the symbols are logically linked together by using the OR operator). In the following example, the presence of either `A` or `B` results in a method call:

C#

```
[Conditional("A"), Conditional("B")]
static void DoIfAorB()
{
    // ...
}
```

Using `Conditional` with attribute classes

The `Conditional` attribute can also be applied to an attribute class definition. In the following example, the custom attribute `Documentation` will only add information to the metadata if `DEBUG` is defined.

C#

```
[Conditional("DEBUG")]
public class DocumentationAttribute : System.Attribute
```

```
{
    string text;

    public DocumentationAttribute(string text)
    {
        this.text = text;
    }
}

class SampleClass
{
    // This attribute will only be included if DEBUG is defined.
    [Documentation("This method displays an integer.")]
    static void DoWork(int i)
    {
        System.Console.WriteLine(i.ToString());
    }
}
```

`Obsolete` attribute

The `Obsolete` attribute marks a code element as no longer recommended for use. Use of an entity marked obsolete generates a warning or an error. The `Obsolete` attribute is a single-use attribute and can be applied to any entity that allows attributes. `Obsolete` is an alias for ObsoleteAttribute.

In the following example, the `Obsolete` attribute is applied to class `A` and to method `B.OldMethod`. Because the second argument of the attribute constructor applied to `B.OldMethod` is set to `true`, this method will cause a compiler error, whereas using class `A` will just produce a warning. Calling `B.NewMethod`, however, produces no warning or error. For example, when you use it with the previous definitions, the following code generates two warnings and one error:

C#

```
namespace AttributeExamples
{
    [Obsolete("use class B")]
    public class A
    {
        public void Method() { }
    }

    public class B
    {
        [Obsolete("use NewMethod", true)]
        public void OldMethod() { }
        public void NewMethod() { }
    }

    public static class ObsoleteProgram
    {
        public static void Main()
        {
            // Generates 2 warnings:
            A a = new A();

            // Generate no errors or warnings:
            B b = new B();
            b.NewMethod();

            // Generates an error, compilation fails.
            // b.OldMethod();
        }
    }
}
```

The string provided as the first argument to the attribute constructor will be displayed as part of the warning or error. Two warnings for class `A` are generated: one for the declaration of the class reference, and one for the class constructor. The `Obsolete` attribute can be used without arguments, but including an explanation what to use instead is recommended.

In C# 10, you can use constant string interpolation and the `nameof` operator to ensure the names match:

```C#
public class B
{
    [Obsolete($"use {nameof(NewMethod)} instead", true)]
    public void OldMethod() { }

    public void NewMethod() { }
}
```

`SetsRequiredMembers` attribute

The `SetsRequiredMembers` attribute informs the compiler that a constructor sets all `required` members in that class or struct. The compiler assumes any constructor with the System.Diagnostics.CodeAnalysis.SetsRequiredMembersAttribute attribute initializes all `required` members. Any code that invokes such a constructor doesn't need object initializers to set required members. This is primarily useful for positional records and primary constructors.

`AttributeUsage` attribute

The `AttributeUsage` attribute determines how a custom attribute class can be used. AttributeUsageAttribute is an attribute you apply to custom attribute definitions. The `AttributeUsage` attribute enables you to control:

- Which program elements attribute may be applied to. Unless you restrict its usage, an attribute may be applied to any of the following program elements:
  - Assembly
  - Module
  - Field
  - Event
  - Method
  - Param
  - Property
  - Return
  - Type
- Whether an attribute can be applied to a single program element multiple times.
- Whether attributes are inherited by derived classes.

The default settings look like the following example when applied explicitly:

C#

```
[AttributeUsage(AttributeTargets.All,
                   AllowMultiple = false,
                   Inherited = true)]
class NewAttribute : Attribute { }
```

In this example, the `NewAttribute` class can be applied to any supported program element. But it can be applied only once to each entity. The attribute is inherited by derived classes when applied to a base class.

The AllowMultiple and Inherited arguments are optional, so the following code has the same effect:

C#

```
[AttributeUsage(AttributeTargets.All)]
```

```
class NewAttribute : Attribute { }
```

The first AttributeUsageAttribute argument must be one or more elements of the AttributeTargets enumeration. Multiple target types can be linked together with the OR operator, like the following example shows:

C#

```
[AttributeUsage(AttributeTargets.Property | AttributeTargets.Field)]
class NewPropertyOrFieldAttribute : Attribute { }
```

Attributes can be applied to either the property or the backing field for an auto-implemented property. The attribute applies to the property, unless you specify the `field` specifier on the attribute. Both are shown in the following example:

C#

```
class MyClass
{
    // Attribute attached to property:
    [NewPropertyOrField]
    public string Name { get; set; } = string.Empty;

    // Attribute attached to backing field:
    [field: NewPropertyOrField]
    public string Description { get; set; } = string.Empty;
}
```

If the AllowMultiple argument is `true`, then the resulting attribute can be applied more than once to a single entity, as shown in the following example:

C#

```
[AttributeUsage(AttributeTargets.Class, AllowMultiple = true)]
class MultiUse : Attribute { }

[MultiUse]
[MultiUse]
class Class1 { }

[MultiUse, MultiUse]
class Class2 { }
```

In this case, `MultiUseAttribute` can be applied repeatedly because `AllowMultiple` is set to `true`. Both formats shown for applying multiple attributes are valid.

If Inherited is `false`, then the attribute isn't inherited by classes derived from an attributed class. For example:

```C#
[AttributeUsage(AttributeTargets.Class, Inherited = false)]
class NonInheritedAttribute : Attribute { }

[NonInherited]
class BClass { }

class DClass : BClass { }
```

In this case `NonInheritedAttribute` isn't applied to `DClass` via inheritance.

You can also use these keywords to specify where an attribute should be applied. For example, you can use the `field:` specifier to add an attribute to the backing field of an auto-implemented property. Or you can use the `field:`, `property:` or `param:` specifier to apply an attribute to any of the elements generated from a positional record. For an example, see Positional syntax for property definition.

`AsyncMethodBuilder` attribute

You add the System.Runtime.CompilerServices.AsyncMethodBuilderAttribute attribute to a type that can be an async return type. The attribute specifies the type that builds the async method implementation when the specified type is returned from an async method. The `AsyncMethodBuilder` attribute can be applied to a type that:

- Has an accessible `GetAwaiter` method.
- The object returned by the `GetAwaiter` method implements the System.Runtime.CompilerServices.ICriticalNotifyCompletion interface.

The constructor to the `AsyncMethodBuilder` attribute specifies the type of the associated builder. The builder must implement the following accessible members:

- A static `Create()` method that returns the type of the builder.
- A readable `Task` property that returns the async return type.
- A `void SetException(Exception)` method that sets the exception when a task faults.
- A `void SetResult()` or `void SetResult(T result)` method that marks the task as completed and optionally sets the task's result

- A `Start` method with the following API signature:

  C#

  ```
  void Start<TStateMachine>(ref TStateMachine stateMachine)
            where TStateMachine :
  System.Runtime.CompilerServices.IAsyncStateMachine
  ```

- An `AwaitOnCompleted` method with the following signature:

  C#

  ```
  public void AwaitOnCompleted<TAwaiter, TStateMachine>(ref
  TAwaiter awaiter, ref TStateMachine stateMachine)
      where TAwaiter :
  System.Runtime.CompilerServices.INotifyCompletion
      where TStateMachine :
  System.Runtime.CompilerServices.IAsyncStateMachine
  ```

- An `AwaitUnsafeOnCompleted` method with the following signature:

  C#

  ```
        public void AwaitUnsafeOnCompleted<TAwaiter, TStateMachine>
  (ref TAwaiter awaiter, ref TStateMachine stateMachine)
            where TAwaiter :
  System.Runtime.CompilerServices.ICriticalNotifyCompletion
            where TStateMachine :
  System.Runtime.CompilerServices.IAsyncStateMachine
  ```

You can learn about async method builders by reading about the following builders supplied by .NET:

- System.Runtime.CompilerServices.AsyncTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncTaskMethodBuilder<TResult>
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder
- System.Runtime.CompilerServices.AsyncValueTaskMethodBuilder<TResult>

In C# 10 and later, the `AsyncMethodBuilder` attribute can be applied to an async method to override the builder for that type.

`InterpolatedStringHandler` and `InterpolatedStringHandlerArguments` attributes

Starting with C# 10, you use these attributes to specify that a type is an *interpolated string handler*. The .NET 6 library already includes System.Runtime.CompilerServices.DefaultInterpolatedStringHandler for scenarios where you use an interpolated string as the argument for a `string` parameter. You may have other instances where you want to control how interpolated strings are processed. You apply the System.Runtime.CompilerServices.InterpolatedStringHandlerAttribute to the type that implements your handler. You apply the System.Runtime.CompilerServices.InterpolatedStringHandlerArgumentAttribute to parameters of that type's constructor.

You can learn more about building an interpolated string handler in the C# 10 feature specification for interpolated string improvements.

`ModuleInitializer` attribute

Beginning with C# 9, the `ModuleInitializer` attribute marks a method that the runtime calls when the assembly loads. `ModuleInitializer` is an alias for ModuleInitializerAttribute.

The `ModuleInitializer` attribute can only be applied to a method that:

- Is static.
- Is parameterless.
- Returns `void`.
- Is accessible from the containing module, that is, `internal` or `public`.
- Isn't a generic method.
- Isn't contained in a generic class.
- Isn't a local function.

The `ModuleInitializer` attribute can be applied to multiple methods. In that case, the order in which the runtime calls them is deterministic but not specified.

The following example illustrates use of multiple module initializer methods. The `Init1` and `Init2` methods run before `Main`, and each adds a string to the `Text` property. So when `Main` runs, the `Text` property already has strings from both initializer methods.

C#

```
using System;

internal class ModuleInitializerExampleMain
{
    public static void Main()
    {
        Console.WriteLine(ModuleInitializerExampleModule.Text);
        //output: Hello from Init1! Hello from Init2!
    }
}
```

C#

```csharp
using System.Runtime.CompilerServices;

internal class ModuleInitializerExampleModule
{
    public static string? Text { get; set; }

    [ModuleInitializer]
    public static void Init1()
    {
        Text += "Hello from Init1! ";
    }

    [ModuleInitializer]
    public static void Init2()
    {
        Text += "Hello from Init2! ";
    }
}
```

Source code generators sometimes need to generate initialization code. Module initializers provide a standard place for that code. In most other cases, you should write a static constructor instead of a module initializer.

`SkipLocalsInit` attribute

Beginning in C# 9, the `SkipLocalsInit` attribute prevents the compiler from setting the `.locals init` flag when emitting to metadata. The `SkipLocalsInit` attribute is a single-use attribute and can be applied to a method, a property, a class, a struct, an interface, or a module, but not to an assembly. `SkipLocalsInit` is an alias for SkipLocalsInitAttribute.

The `.locals init` flag causes the CLR to initialize all of the local variables declared in a method to their default values. Since the compiler also makes sure that you never use a variable before assigning some value to it, `.locals init` is typically not necessary. However, the extra zero-initialization may have measurable performance impact in some scenarios, such as when you use stackalloc to allocate an array on the stack. In those cases, you can add the `SkipLocalsInit` attribute. If applied to a method directly, the attribute affects that method and all its nested functions, including lambdas and local functions. If applied to a type or module, it affects all methods nested inside. This attribute doesn't affect abstract methods, but it does affect code generated for the implementation.

This attribute requires the AllowUnsafeBlocks compiler option. This requirement signals that in some cases code could view unassigned memory (for example, reading from uninitialized stack-allocated memory).

The following example illustrates the effect of `SkipLocalsInit` attribute on a method that uses `stackalloc`. The method displays whatever was in memory when the array of integers was allocated.

C#

```csharp
[SkipLocalsInit]
static void ReadUninitializedMemory()
{
    Span<int> numbers = stackalloc int[120];
    for (int i = 0; i < 120; i++)
    {
        Console.WriteLine(numbers[i]);
    }
}
// output depends on initial contents of memory, for example:
//0
//0
//0
//168
//0
//-1271631451
//32767
//38
//0
//0
//0
//38
// Remaining rows omitted for brevity.
```

To try this code yourself, set the `AllowUnsafeBlocks` compiler option in your *.csproj* file:

XML

```xml
<PropertyGroup>
  ...
  <AllowUnsafeBlocks>true</AllowUnsafeBlocks>
</PropertyGroup>
```

See also

- Attribute
- System.Reflection
- Attributes
- Reflection