Custom Attributes in .NET

Posted by **Code Maze** | Updated Date Mar 8, 2022 | **2**

Ready to take your skills to the next level? Jump into our high-impact courses in web development and software architecture, all with a focus on mastering the .NET/C# framework. Whether you're **building sleek web applications or designing scalable software solutions**, our expert-led training will give you the tools to succeed. Visit our **COURSES** page now and kickstart your journey!

Custom attributes in .NET (Core) are a helpful mechanism to attach additional information to classes, structs, and even their members. In this article, we're going to explain how to create, access, and get the information from custom attributes in .NET through some practical examples.

> To download the source code for this article, you can visit our **GitHub repository**.

Let's start.

Declaring Custom A

We can define an attribute by creating a cla

Microsoft recommends appending the *'Attribute'* suffix to the end of the class's name. After that, each property of our derived class will be a parameter of the desired data type.

Customizing Custom Attribute Usage

The `AttributeUsageAttribute` class specifies the usage of another attribute class by defining some of the fundamental features.

This class has three members:

- AttributeTargets enum
- Inherited property (bool)
- AllowMultiple property (bool)

AttributeTargets Enum

The `AttributeTargets` enum specifies application elements we can apply our custom attribute to.

To see how it works, let's create a new class called `TaskDescriptorAttribute`:

```
[AttributeUsage(AttributeTargets.Class | AttributeTargets.Struct)]
public class TaskDescriptorAttribute : Attribute
{
    public string? Name { get; set; }
    public string? Description { get; set; }
    public bool NeedsManager { get; set; }
    public int DeveloperCount { get; set; }
}
```

We can apply this attribute (`TaskDescriptorAttribute`) to the classes and structures because we have set its target to both using bitwise combination. Of course, other than classes and structs, we can use methods, enums, and other application elements when creating attributes. Those attributes also apply to all of the application elements if we use `AttributeTargets.All` value (it's the default).

Let's use our attribute with the `MyTasks` class:

```
[TaskDescriptor(Name = "The task's name",
    Description = "Some descriptions for the task",
    NeedsManager = true,
    DeveloperCount = 5)]
public class MyTasks
{
}
```

When we apply our `TaskDescriptorAttribute` to a class, we just use the `TaskDescriptor` part because the compiler lets us use it without the 'Attribute' suffix.

AllowMultiple Property

The `AllowMultiple` property permits multiple instances of our attribute. This property can be either `false` (the default) or `true`.

Let's create another attribute called `DeveloperTaskAttribute`:

```
[AttributeUsage(AttributeTargets.Method, AllowMultiple = true)]
public class DeveloperTaskAttribute : Attribute
{
    public Priorities Priority { get; set; }
    public string? Description { get; set; }
    public DeveloperTaskAttribute(Priorities priority)
    {
        Priority = priority;
    }
}
```

We can use this attribute only on methods. And we can apply multiple instances of it to them. It has the **required** `Priority` and the **optional** `Description` parameters.

To apply this attribute, we are going to create a new `ScheduleMeeting()` method inside the `MyTasks` class:

```
public class MyTasks
{
    [DeveloperTask(Priorities.Low)]
    [DeveloperTask(Priorities.High, Description = "High level description")]
    public void ScheduleMeeting()
    {
    }
}
```

The `ScheduleMeeting()` method has two `DeveloperTask` attributes now. We declare the first one with only the required parameter but the second one with both the required and optional parameters. We cannot define a `DeveloperTask` attribute without a `Priorities` parameter though, or we'll get a compiler error.

Inherited Property

The `Inherited` property is another key feature that we can apply to a custom attribute. It indicates whether that attribute can be inherited. This property has a default value of `true`.

To see the usage of this property, let's create the `ManagerTaskAttribute` attribute:

```
[AttributeUsage(AttributeTargets.Method, Inherited = false)]
public class ManagerTaskAttribute : Attribute
{
    public Priorities Priority { get; set; }
    public bool NeedsReport { get; set; }
}
```

This new attribute cannot be inherited, and both of its parameters are optional.

Now, we are going to create the `ScheduleInterview` method in the `MyTasks` class to utilize it:

```
public class MyTasks
{
    [ManagerTask(Priority = Priorities.Mid, NeedsReport = true)]
    [DeveloperTask(Priorities.High, Description = "High level description")]
    public virtual void ScheduleInterview()
    {
    }
}
```

This method has two attributes, one `DeveloperTask` and one `ManagerTask` attribute each. We also add the `virtual` keyword because we want to override it in another class.

So, let's create the `YourTasks` class that inherits from the `MyTasks` class:

```
public class YourTasks : MyTasks
{
    [DeveloperTask(Priorities.Mid, Description = "Mid level description")]
    public override void ScheduleInterview()
    {
    }
}
```

The `ScheduleInterview` method inside the `YourTasks` inherited class overrides the previous `ScheduleInterview` method from the `MyTasks` base class. This method does not have the `ManagerTask` attribute because its `Inherited` property has a value of `false`.

But, the `DeveloperTask` attribute has a default `Inherited` value of `true`. So, the `YourTasks.ScheduleInterview` method has two `DeveloperTask` attributes. We have declared the first one inside the `YourTasks` class and the second one inside the `MyTasks` class.

Access an Instance of a Custom Attribute

Once we want to retrieve the values from our attributes, we can use the static `GetCustomAttribute` method from the `Attribute` class. So, let's create the `GetAttribute` method for acquiring the information stored in the `TaskDescriptor` instance:

```
public static string? GetAttribute(Type desiredType, Type desiredAttribute)
{
    var attributeInstance = Attribute.GetCustomAttribute(desiredType, desiredAttribute);

    if (attributeInstance == null)
        Console.WriteLine($"The class {desiredType} does not have atributes.");
    else
        WriteOnTheConsole(attributeInstance);

    return attributeInstance?.ToString();
}
```

Our `GetAttribute` method expects a (class) type and an attribute type as input parameters and prints the information on the console.

Inside the `Attribute` base class, we can find different overloads of the `GetCustomAttribute` method. For our example, we use the `GetCustomAttribute(MemberInfo element, Type attributeType)` overload to get our desired information.

The `Type` inherits from the `MemberInfo` base class so, we can pass it to the method as the first argument. We also send our custom attribute's type as the second argument. We print all information on the console using a bit of reflection inside the `WriteOnTheConsole()` method. (you can check the **source code for implementation**)

Creating an Instance of a Custom Attribute

The `GetCustomAttribute` method returns either an instance of the attribute or a `null` value. So, the `attributeInstance` variable stores an instance of our custom attribute if it does exist. Now, we are going to retrieve the information of that instance.

Retrieving the Information of a Custom Attribute

To retrieve the information of our custom attribute, we are going to call the `GetAttribute` method and pass `typeof(MyTasks)` and `typeof(TaskDescriptorAttribute)` as its arguments:

```
GetAttribute(typeof(MyTasks), typeof(TaskDescriptorAttribute));
```

If the `GetAttribute` method finds an instance of the `TaskDescriptorAttribute` class, we should get all of its properties as a result:

```
The CustomAttributes.TaskDescriptorAttribute attribute:
The Name property is: The task's name
The Description property is: Some descriptions for the task
The NeedsManager property is: True
The DeveloperCount property is: 5
```

We retrieved the information of a custom attribute class successfully.

Getting the Instances of Different Custom Attributes

Sometimes, we need to access all attributes of the class's members. The `Attribute` base class has another `GetCustomAttributes` method to return them as an array.

Let's create the `GetAttributesOfMethods` method to access instances of all attributes and retrieve their information:

```
public static List<string> GetAttributesOfMethods(Type elementType)
{
    List<string> attributes = new List<string>();

    var methodInfoList = elementType.GetMethods(BindingFlags.Public |
        BindingFlags.Instance |
        BindingFlags.DeclaredOnly);

    if (methodInfoList == null || methodInfoList.Length == 0)
    {
        Console.WriteLine($"The type {elementType} does not have any methods.");
        return attributes;
    }

    foreach (var methodInfo in methodInfoList)
    {
        var attributeList = Attribute.GetCustomAttributes(methodInfo, true);

        if (attributeList.Length == 0)
        {
            Console.WriteLine($"The {elementType.Name}.{methodInfo.Name} method does not have attributes.");
            continue;
        }

        Console.WriteLine($"The {elementType.Name}.{methodInfo.Name} method's attribute:");

        foreach (var att in attributeList)
        {
            WriteOnTheConsole(att);
            attributes.Add(methodInfo.Name + "-" + att.ToString());
        }

        Console.WriteLine();
    }

    return attributes;
}
```

We want to obtain all of the attributes of every declared method. So, we call the `GetMethods` method using appropriated enums.

Inside the first `foreach` loop, we call the `GetCustomAttributes` method using the `GetCustomAttribute(MemberInfo element, bool inherit)` overload and send the fetched `MethodInfo` values, one by one. We also pass `true` as the second argument because we need the inherited attributes.

Inside the inner `foreach` loop, we sequentially fetch items of the `attributeList` array and print them on the console using the `WriteOnTheConsole` method.

Now, let's call the `GetAttributesOfMethods` method for the `MyTasks` class:

```
GetAttributesOfMethods(typeof(MyTasks));
```

We send the type of the `MyTasks` class to this method and as a result, we expect to see all the attributes of its methods:

```
The MyTasks.ScheduleMeeting method's attribute:

The CustomAttributes.DeveloperTaskAttribute attribute:
The Description property is:
The Priority property is: Low

The CustomAttributes.DeveloperTaskAttribute attribute:
The Description property is: High level description
The Priority property is: High

The MyTasks.ScheduleInterview method's attribute:

The CustomAttributes.ManagerTaskAttribute attribute:
The Priority property is: Mid
The NeedsReport property is: True

The CustomAttributes.DeveloperTaskAttribute attribute:
The Description property is: High level description
The Priority property is: High
```

The `GetAttributesOfMethods` prints the attributes of `ScheduleMeeting` and `ScheduleInterview` methods respectively.

We can do the same for the `YourTasks` class:

```
GetAttributesOfMethods(typeof(YourTasks));
```

And find the results very similar:

```
The YourTasks.ScheduleInterview method's attribute:

The CustomAttributes.DeveloperTaskAttribute attribute:
The Description property is: Mid level description
The Priority property is: Mid

The CustomAttributes.DeveloperTaskAttribute attribute:
The Description property is: High level description
The Priority property is: High
```

We don't see the `ManagerTask` attribute on the console because this attribute has an `Inherited` value of `false`. The reason we see two `DeveloperTask` attributes is because we've declared the first one inside the `MyTasks` class and the second inside the `YourTasks` class.

Conclusion

In this article, we have learned how to declare custom attributes in .NET. We've seen that we can utilize them for classes and their members. We also have touched upon the access ways to the single and multiple attribute instances. Finally, we have found out how to retrieve their information.