**C# Indexers and Properties**

  Table of Contents

Indexers enable a feature by which an object of a class can be indexed like an array, whereas properties enable to manage access to a class instance data.

As you go through this article, you will come to know that, indexers and properties are related to each other, because both are dependent upon the C# accessor feature. At the end of this article, you will find the differences between indexers and properties.

C# Indexers:

An indexer is a smart array that enables an instance of a class or structure to be indexed like an array. Indexers must have at least one parameter else a compile-time exception will be generated. It is defined with this keyword and parameters, otherwise, it is the same as property. The syntax for one-dimensional indexer is shown here:

<access\_modifier> <return\_type> this[parameters]

{

get

{

// return the value specified by index

}

set

{

// set a new value

}

}

Let's see the brief description of each term

[**access\_modifier**](https://www.studytonight.com/post/csharp-access-modifiers)**:** It can be **public**, **private**, **protected** or **internal**.

**return\_type:** It can be any valid [C# data type](https://www.studytonight.com/post/csharp-variables-and-data-types)

**this:** It is the keyword which points to the object of the current class

**parameters:** This specifies the parameter list of the indexer

**get** and **set**: These are the accessors

Let's see an implementation of an indexer,

**Filename:** Program.cs

using System;

namespace Studytonight

{

class Employee

{

private string[] arr = new string[2];

public string this[int i]

{

get

{

return arr[i];

}

set

{

arr[i] = value;

}

}

}

public class Program

{

public static void Main(string[] args)

{

Employee eid = new Employee();

eid[0] = "NOBANK129394940";

eid[1] = "NOBANK129394941";

for(int i=0; i<2; i++)

{

Console.WriteLine(eid[i]);

}

}

}

}

**Output:**

NOBANK129394940

NOBANK129394941

In the above example, we have implemented an indexer in the **Employee** class. The class contains a private string array that external users can’t see. The indexer has a get and a set accessor method, where the get method is used to return indexer value whereas, the set method is used to assign a new value. In the set accessor, the value is a keyword through which the value gets assigned. Here, the Employee class behaves like a virtual array, and we accessed the class instance values using an array access operator ([ ]).

C# Properties:

The properties are also called as accessors as it is a member function of the class and provides a feature through which the values of a private field can be read, written, or changed. The property has two accessors, get and set, and these accessors helps to change the implementation of the class variables. The syntax for a property is shown here:

<access\_modifier> <return\_type> <property\_name>

{

get

{

// return the property value

}

set

{

// set a new value

}

}

Copy

Let's see one practical example of properties

**Filename:** Program.cs

using System;

namespace Studytonight

{

class Employee

{

private string eid = "NOBANK";

public string Emp

{

get

{

return eid;

}

set

{

eid = value;

}

}

}

public class Program

{

public static void Main(string[] args)

{

Employee eid1 = new Employee();

eid1.Emp = "NOBANK129394940";

Console.WriteLine(eid1.Emp);

}

}

}

Copy

**Output:**

NOBANK129394940

In the above example, the eid variable is marked as private, and we are changing/assigning a new value using the property Emp. The Emp property has two accessors, get and set, the get accessor is used to return the property value, whereas set is used to assign a new value. In the main method, we have created the object of Employee class, and with the help of object and property Emp, we assigned the value.

Difference between Indexers and Properties:

| **Indexers** | **Properties** |
| --- | --- |
| Indexers are created using this keyword | Properties don't require this keyword |
| Indexers are declared with at least one parameter | Properties are always declared without parameters |
| Indexers are accessed/invoked using indexes | Properties are accessed/invoked using the specified name |
| Indexers can't be static | Properties can be declared as static |
| Indexers can be overloaded | Properties can't be overloaded |

Conclusion:

We hope this article helped you to understand the indexers and properties in C# language. If you have any queries regarding this topic, then please let us know in the comment section below. We are happy to solve your doubts.

//=============================================================

**Custom Attributes :**

Custom attributes in C# provide a powerful mechanism for extending the metadata and runtime behaviors of code elements. By adding custom attributes to classes, methods, properties, and fields, developers can enhance the information associated with these elements and influence their behavior at runtime.

This article dives into the world of custom attributes in C#, exploring their significance, how to define them, and how to apply them to various code elements. Additionally, it delves into techniques for retrieving and utilizing custom attributes at runtime, leveraging them for enhanced reflection capabilities. Finally, it offers best practices for creating and using custom attributes, ensuring their effective and efficient utilization in C# projects.

Introduction to Custom Attributes in C#

**Custom attributes** in C# are a way to extend metadata and add additional information to your code elements. They allow you to attach custom metadata to classes, methods, properties, fields, and more.

There are inbuilt attributes in C# that you can use to add metadata to your code elements. For example, the [Serializable] attribute can be used to mark a class as serializable. However, you can also create your own custom attributes to add metadata or configure runtime behaviors.

Custom attributes play a crucial role in enhancing the functionality and behavior of your code. They provide a flexible mechanism for adding descriptive information, configuring runtime behaviors, and enabling runtime reflection.

Custom attributes in C# are implemented as classes that derive from the Attribute base class. These attribute classes can then be applied to various code elements to provide additional metadata or configure behaviors.

Creating custom attribute classes

To create a custom attribute, you define a class that inherits from the Attribute base class. You can add properties, fields, and methods to the attribute class to store and manipulate data. Once defined, you can use the custom attribute by applying it to code elements using square brackets.

AttributeUsage Attribute

The AttributeUsage attribute allows you to specify how your custom attribute can be used. It allows you to control which code elements can be decorated with your custom attribute, how many times it can be applied, and whether it can be inherited by derived classes.

The AttributeUsage attribute is applied to your custom attribute class. It takes three parameters:

1. **AttributeTargets**: Specifies the code elements that can be decorated with your custom attribute. This is a required parameter.

|  |  |
| --- | --- |
| 1 | [AttributeUsage(AttributeTargets.All)] |

AttributeTargets is an enumeration that contains the following values:

* All: Specifies that your custom attribute can be applied to any code element.
* Assembly: Specifies that your custom attribute can be applied to an assembly.
* Class: Specifies that your custom attribute can be applied to a class.
* Method: Specifies that your custom attribute can be applied to a method.
* Module: Specifies that your custom attribute can be applied to a module.
* Constructor: Specifies that your custom attribute can be applied to a constructor.
* Delegate: Specifies that your custom attribute can be applied to a delegate.
* Enum: Specifies that your custom attribute can be applied to an enumeration.
* Event: Specifies that your custom attribute can be applied to an event.
* Field: Specifies that your custom attribute can be applied to a field.
* GenericParameter: Specifies that your custom attribute can be applied to a generic parameter.
* Interface: Specifies that your custom attribute can be applied to an interface.
* Parameter: Specifies that your custom attribute can be applied to a parameter.
* Property: Specifies that your custom attribute can be applied to a property.

These are just a few of the values available in the AttributeTargets enumeration. For a full list, see the AttributeTargets documentation.

1. **AllowMultiple**: Specifies whether your custom attribute can be applied multiple times to the same code element. This is an optional parameter that defaults to false.

|  |  |
| --- | --- |
| 1 | [AttributeUsage(AttributeTargets.All, AllowMultiple = true)] |

1. **Inherited**: Specifies whether your custom attribute can be inherited by derived classes. This is an optional parameter that defaults to false.

|  |  |
| --- | --- |
| 1 | [AttributeUsage(AttributeTargets.All, Inherited = true)] |

Defining Attribute Class

|  |  |
| --- | --- |
| 1  2  3  4  5 | [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true)]  public class MyCustomAttribute : Attribute  {  // Attribute class implementation  } |

In the above example, we’ve defined a custom attribute class called MyCustomAttribute. It inherits from the Attribute base class and is decorated with the AttributeUsage attribute. This allows the custom attribute to be applied to classes and methods, and it can be applied multiple times to the same code element.

Adding Properties to Attribute Class

|  |  |
| --- | --- |
| 1  2  3  4  5  6 | [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true)]  public class MyCustomAttribute : Attribute  {  public string Name { get; set; }  public int Age { get; set; }  } |

In the above example, we’ve added two properties to the MyCustomAttribute class. These properties can be used to store data that can be retrieved at runtime using reflection.

Adding Methods to Attribute Class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11 | [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true)]  public class MyCustomAttribute : Attribute  {  public string Name { get; set; }  public int Age { get; set; }  public void Print()  {  Console.WriteLine($"Name: {Name}, Age: {Age}");  }  } |

In the above example, we’ve added a Print() method to the MyCustomAttribute class. This method can be used to perform actions at runtime, such as printing the values of the attribute’s properties.

Adding Constructor to Attribute Class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18 | [AttributeUsage(AttributeTargets.Class | AttributeTargets.Method, AllowMultiple = true)]  public class MyCustomAttribute : Attribute  {  public string Name { get; set; }  public int Age { get; set; }  public void Print()  {  Console.WriteLine($"Name: {Name}, Age: {Age}");  }  public MyCustomAttribute(string name, int age)  {  Name = name;  Age = age;  }  } |

In the above example, we’ve added a constructor to the MyCustomAttribute class. This constructor can be used to initialize the attribute’s properties when it’s applied to a code element.

Using Attribute Class

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9 | [MyCustom("John Doe", 42)]  public class MyClass  {  [MyCustom("Jane Doe", 21)]  public void MyMethod()  {  // Method implementation  }  } |

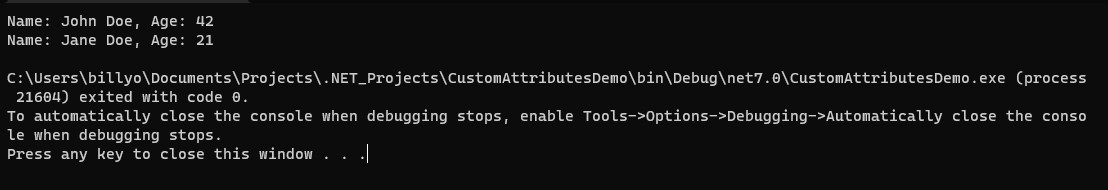
In the above example, we’ve applied the MyCustom attribute to the MyClass class and the MyMethod() method. We’ve also passed in values for the attribute’s properties using the attribute’s constructor.

Using Attribute Class Properties

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | [MyCustom("John Doe", 42)]  public class MyClass  {  [MyCustom("Jane Doe", 21)]  public void MyMethod()  {  // Method implementation  }  }  class Program  {  static void Main(string[] args)  {  var myClass = new MyClass();  var myMethod = myClass.GetType().GetMethod("MyMethod");  var classAttributes = myClass.GetType().GetCustomAttributes();  var methodAttributes = myMethod.GetCustomAttributes();  foreach (MyCustomAttribute attribute in classAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  foreach (MyCustomAttribute attribute in methodAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  }  } |

In the above example, we’ve retrieved the MyCustom attribute from the MyClass class and the MyMethod() method using reflection. We’ve then iterated over the attributes and printed the values of their properties.

Running the above example produces the following output:

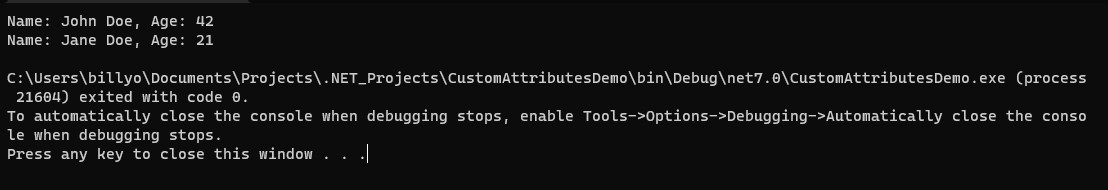
[](https://www.billyokeyo.com/assets/img/custom-attributes/class-constructor.jpg)

Using Attribute Class Methods

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | [MyCustom("John Doe", 42)]  public class MyClass  {  [MyCustom("Jane Doe", 21)]  public void MyMethod()  {  // Method implementation  }  }  class Program  {  static void Main(string[] args)  {  var myClass = new MyClass();  var myMethod = myClass.GetType().GetMethod("MyMethod");  var classAttributes = myClass.GetType().GetCustomAttributes();  var methodAttributes = myMethod.GetCustomAttributes();  foreach (MyCustomAttribute attribute in classAttributes)  {  attribute.Print();  }  foreach (MyCustomAttribute attribute in methodAttributes)  {  attribute.Print();  }  }  } |

In the above example, we’ve retrieved the MyCustom attribute from the MyClass class and the MyMethod() method using reflection. We’ve then iterated over the attributes and called their Print() methods.

Running the above example produces the following output:

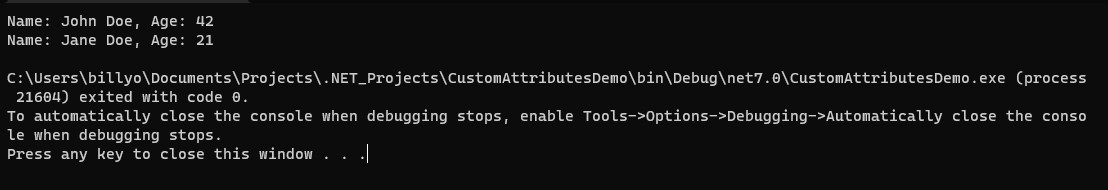
[](https://www.billyokeyo.com/assets/img/custom-attributes/class-constructor.jpg)

Using Attribute Class Constructor

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31 | [MyCustom("John Doe", 42)]  public class MyClass  {  [MyCustom("Jane Doe", 21)]  public void MyMethod()  {  // Method implementation  }  }  class Program  {  static void Main(string[] args)  {  var myClass = new MyClass();  var myMethod = myClass.GetType().GetMethod("MyMethod");  var classAttributes = myClass.GetType().GetCustomAttributes();  var methodAttributes = myMethod.GetCustomAttributes();  foreach (MyCustomAttribute attribute in classAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  foreach (MyCustomAttribute attribute in methodAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  }  } |

In the above example, we’ve retrieved the MyCustom attribute from the MyClass class and the MyMethod() method using reflection. We’ve then iterated over the attributes and printed the values of their Name and Age properties.

Running the above code gives the following output:

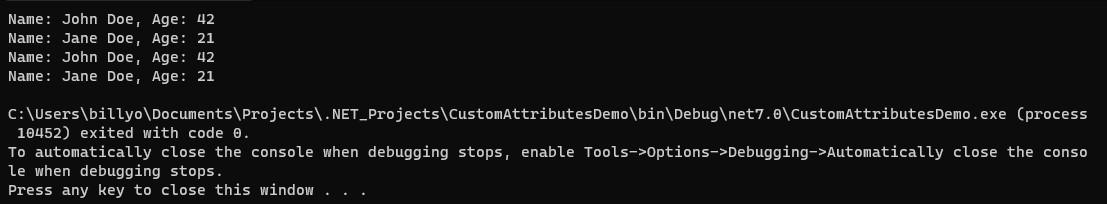
[](https://www.billyokeyo.com/assets/img/custom-attributes/class-constructor.jpg)

Using Attribute Class with Multiple Instances

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33 | [MyCustom("John Doe", 42)]  [MyCustom("Jane Doe", 21)]  public class MyClass  {  [MyCustom("John Doe", 42)]  [MyCustom("Jane Doe", 21)]  public void MyMethod()  {  // Method implementation  }  }  class Program  {  static void Main(string[] args)  {  var myClass = new MyClass();  var myMethod = myClass.GetType().GetMethod("MyMethod");  var classAttributes = myClass.GetType().GetCustomAttributes();  var methodAttributes = myMethod.GetCustomAttributes();  foreach (MyCustomAttribute attribute in classAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  foreach (MyCustomAttribute attribute in methodAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  }  } |

In the above example, we’ve retrieved the MyCustom attribute from the MyClass class and the MyMethod() method using reflection. We’ve then iterated over the attributes and printed the values of their Name and Age properties.

Running the above code will produce the following output:

[](https://www.billyokeyo.com/assets/img/custom-attributes/class-multiple-instances.jpg)

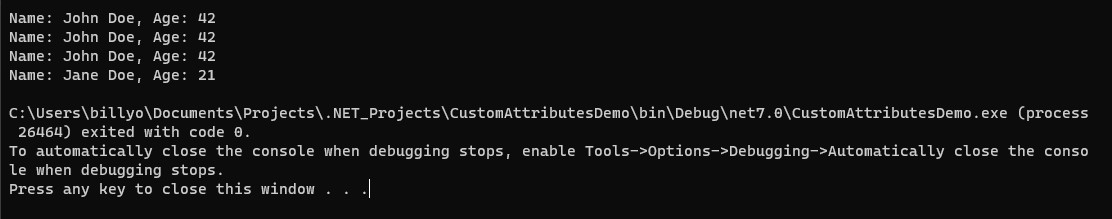
Using Attribute Class with Inheritance

|  |  |
| --- | --- |
| 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30  31  32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56 | [MyCustom("John Doe", 42)]  public class MyClass  {  [MyCustom("John Doe", 42)]  public void MyMethod()  {  // Method implementation  }  }  public class MyDerivedClass : MyClass  {  [MyCustom("Jane Doe", 21)]  public void MyDerivedMethod()  {  // Method implementation  }  }  class Program  {  static void Main(string[] args)  {  var myClass = new MyClass();  var myMethod = myClass.GetType().GetMethod("MyMethod");  var classAttributes = myClass.GetType().GetCustomAttributes();  var methodAttributes = myMethod.GetCustomAttributes();  foreach (MyCustomAttribute attribute in classAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  foreach (MyCustomAttribute attribute in methodAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  var myDerivedClass = new MyDerivedClass();  var myDerivedMethod = myDerivedClass.GetType().GetMethod("MyDerivedMethod");  var derivedClassAttributes = myDerivedClass.GetType().GetCustomAttributes();  var derivedMethodAttributes = myDerivedMethod.GetCustomAttributes();  foreach (MyCustomAttribute attribute in derivedClassAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  foreach (MyCustomAttribute attribute in derivedMethodAttributes)  {  Console.WriteLine($"Name: {attribute.Name}, Age: {attribute.Age}");  }  }  } |

In the above example, we’ve retrieved the MyCustom attribute from the MyClass class and the MyMethod() method using reflection. We’ve then iterated over the attributes and printed the values of their Name and Age properties.

We’ve also created a derived class, MyDerivedClass, which inherits from MyClass. We’ve then retrieved the MyCustom attribute from the MyDerivedClass class and the MyDerivedMethod() method using reflection. We’ve then iterated over the attributes and printed the values of their Name and Age properties.

Running the above code will produce the following output:

[](https://www.billyokeyo.com/assets/img/custom-attributes/inheritance.jpg)

In conclusion, custom attributes in C# provide a valuable tool for extending metadata and runtime behaviors in code. By creating and applying custom attributes, developers can enhance the information associated with their code elements, influencing their behavior at runtime. Leveraging custom attributes for reflection opens up new possibilities for dynamic programming and advanced introspection. However, it is important to follow best practices when creating and using custom attributes to ensure their effectiveness and maintainability. As the C# language continues to evolve, it is worth considering future developments in custom attribute usage and how they can further enhance the capabilities of our applications.