Properties (C# Programming Guide)

03/10/2017 • 4 minutes to read • +7

In this article

A property is a member that provides a flexible mechanism to read, write, or compute the value of a private field. Properties can be used as if they are public data members, but they are actually special methods called *accessors*. This enables data to be accessed easily and still helps promote the safety and flexibility of methods.

Properties overview

- Properties enable a class to expose a public way of getting and setting values, while hiding implementation or verification code.
- A get property accessor is used to return the property value, and a set property accessor is used to assign a new value. These accessors can have different access levels. For more information, see Restricting Accessor Accessibility.
- The value keyword is used to define the value being assigned by the `set` accessor.

- Properties can be *read-write* (they have both a `get` and a `set` accessor), *read-only* (they have a `get` accessor but no `set` accessor), or *write-only* (they have a `set` accessor, but no `get` accessor). Write-only properties are rare and are most commonly used to restrict access to sensitive data.

- Simple properties that require no custom accessor code can be implemented either as expression body definitions or as auto-implemented properties.

Properties with backing fields

One basic pattern for implementing a property involves using a private backing field for setting and retrieving the property value. The `get` accessor returns the value of the private field, and the `set` accessor may perform some data validation before assigning a value to the private field. Both accessors may also perform some conversion or computation on the data before it is stored or returned.

The following example illustrates this pattern. In this example, the `TimePeriod` class represents an interval of time. Internally, the class stores the time interval in seconds in a private field named `_seconds`. A read-write property named `Hours` allows the customer to specify the time interval in hours. Both the `get` and the `set` accessors perform the necessary conversion between hours and seconds. In addition, the `set` accessor validates the data and throws an ArgumentOutOfRangeException if the number of hours is invalid.

C# Copy

```
using System;

class TimePeriod
{
   private double _seconds;

   public double Hours
   {
       get { return _seconds / 3600; }
       set {
```

```
            if (value < 0 || value > 24)
               throw new ArgumentOutOfRangeException(
                     $"{nameof(value)} must be between 0 and 24.");

            _seconds = value * 3600;
         }
      }
   }

   class Program
   {
      static void Main()
      {
          TimePeriod t = new TimePeriod();
          // The property assignment causes the 'set' accessor to be called.
          t.Hours = 24;

          // Retrieving the property causes the 'get' accessor to be called.
          Console.WriteLine($"Time in hours: {t.Hours}");
      }
   }
   // The example displays the following output:
   //    Time in hours: 24
```

Expression body definitions

Property accessors often consist of single-line statements that just assign or return the result of an expression. You can implement these properties as expression-bodied members. Expression body definitions consist of the `=>` symbol followed by the expression to assign to or retrieve from the property.

Starting with C# 6, read-only properties can implement the `get` accessor as an expression-bodied member. In this case, neither the `get` accessor keyword nor the `return` keyword is used. The following example implements the read-only `Name` property as an expression-bodied member.

C# Copy

```
using System;

public class Person
{
   private string _firstName;
   private string _lastName;

   public Person(string first, string last)
   {
      _firstName = first;
      _lastName = last;
   }

   public string Name => $"{_firstName} {_lastName}";
}

public class Example
{
   public static void Main()
   {
      var person = new Person("Magnus", "Hedlund");
      Console.WriteLine(person.Name);
   }
}
// The example displays the following output:
//      Magnus Hedlund
```

Starting with C# 7.0, both the `get` and the `set` accessor can be implemented as expression-bodied members. In this case, the `get` and `set` keywords must be present. The following example illustrates the use of expression body definitions for both accessors. Note that the `return` keyword is not used with the `get` accessor.

C# Copy

```
using System;

public class SaleItem
{
   string _name;
   decimal _cost;

   public SaleItem(string name, decimal cost)
   {
      _name = name;
      _cost = cost;
   }

   public string Name
   {
      get => _name;
      set => _name = value;
   }

   public decimal Price
   {
      get => _cost;
      set => _cost = value;
   }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem("Shoes", 19.95m);
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Auto-implemented properties

In some cases, property `get` and `set` accessors just assign a value to or retrieve a value from a backing field without including any additional logic. By using auto-implemented properties, you can simplify your code while having the C# compiler transparently provide the backing field for you.

If a property has both a `get` and a `set` accessor, both must be auto-implemented. You define an auto-implemented property by using the `get` and `set` keywords without providing any implementation. The following example repeats the previous one, except that `Name` and `Price` are auto-implemented properties. Note that the example also removes the parameterized constructor, so that `SaleItem` objects are now initialized with a call to the parameterless constructor and an object initializer.

C# Copy

```
using System;

public class SaleItem
{
   public string Name
   { get; set; }

   public decimal Price
   { get; set; }
}

class Program
{
   static void Main(string[] args)
   {
      var item = new SaleItem{ Name = "Shoes", Price = 19.95m };
      Console.WriteLine($"{item.Name}: sells for {item.Price:C2}");
   }
}
// The example displays output like the following:
//       Shoes: sells for $19.95
```

Related sections

- Using Properties
- Interface Properties
- Comparison Between Properties and Indexers
- Restricting Accessor Accessibility
- Auto-Implemented Properties

C# Language Specification

For more information, see Properties in the C# Language Specification. The language specification is the definitive source for C# syntax and usage.

See also

- C# Programming Guide
- Using Properties
- Indexers
- get keyword
- set keyword