

# Using Properties (C# Programming Guide)

07/20/2015 • 8 minutes to read •  +7

## In this article

[The get Accessor](#)

[The set Accessor](#)

[Remarks](#)

[Example](#)

[Example](#)

[Example](#)

[See also](#)

Properties combine aspects of both fields and methods. To the user of an object, a property appears to be a field, accessing the property requires the same syntax. To the implementer of a class, a property is one or two code blocks, representing a [get](#) accessor and/or a [set](#) accessor. The code block for the `get` accessor is executed when the property is read; the code block for the `set` accessor is executed when the property is assigned a new value. A property without a `set` accessor is considered read-only. A property without a `get` accessor is considered write-only. A property that has both accessors is read-write.

Unlike fields, properties are not classified as variables. Therefore, you cannot pass a property as a [ref](#) or [out](#) parameter.

Properties have many uses: they can validate data before allowing a change; they can transparently expose data on a class where that data is actually retrieved from some other source, such as a database; they can take an action when data is changed, such as raising an event, or changing the value of other fields.

Properties are declared in the class block by specifying the access level of the field, followed by the type of the property, followed by the name of the property, and followed by a code block that declares a `get`-accessor and/or a `set` accessor. For example:

C#

 Copy

```
public class Date
{
    private int _month = 7; // Backing store

    public int Month
    {
        get => _month;
        set
        {
            if ((value > 0) && (value < 13))
            {
                _month = value;
            }
        }
    }
}
```

In this example, `Month` is declared as a property so that the `set` accessor can make sure that the `Month` value is set between 1 and 12. The `Month` property uses a private field to track the actual value. The real location of a property's data is often referred to as the property's "backing store." It is common for properties to use private fields as a backing store. The field is marked private in order to make sure that it can only be changed by calling the property. For more information about public and private access restrictions, see [Access Modifiers](#).

Auto-implemented properties provide simplified syntax for simple property declarations. For more information, see [Auto-Implemented Properties](#).

## The get Accessor

The body of the `get` accessor resembles that of a method. It must return a value of the property type. The execution of the `get` accessor is equivalent to reading the value of the field. For example, when you are returning the private variable from the

`get` accessor and optimizations are enabled, the call to the `get` accessor method is inlined by the compiler so there is no method-call overhead. However, a virtual `get` accessor method cannot be inlined because the compiler does not know at compile-time which method may actually be called at run time. The following is a `get` accessor that returns the value of a private field `_name`:

C#



```
class Person
{
    private string _name; // the name field
    public string Name => _name; // the Name property
}
```

When you reference the property, except as the target of an assignment, the `get` accessor is invoked to read the value of the property. For example:

C#



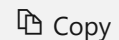
```
Person person = new Person();
//...

System.Console.Write(person.Name); // the get accessor is invoked here
```

The `get` accessor must end in a [return](#) or [throw](#) statement, and control cannot flow off the accessor body.

It is a bad programming style to change the state of the object by using the `get` accessor. For example, the following accessor produces the side effect of changing the state of the object every time that the `_number` field is accessed.

C#



```
private int _number;
public int Number => _number++; // Don't do this
```

The `get` accessor can be used to return the field value or to compute it and return it. For example:

C#

 Copy

```
class Employee
{
    private string _name;
    public string Name => _name != null ? _name : "NA";
}
```

In the previous code segment, if you do not assign a value to the `Name` property, it will return the value `NA`.

## The set Accessor

The `set` accessor resembles a method whose return type is `void`. It uses an implicit parameter called `value`, whose type is the type of the property. In the following example, a `set` accessor is added to the `Name` property:

C#

 Copy

```
class Person
{
    private string _name; // the name field
    public string Name    // the Name property
    {
        get => _name;
        set => _name = value;
    }
}
```

When you assign a value to the property, the `set` accessor is invoked by using an argument that provides the new value. For example:

C#



```
Person person = new Person();  
person.Name = "Joe"; // the set accessor is invoked here  
  
System.Console.Write(person.Name); // the get accessor is invoked here
```

It is an error to use the implicit parameter name, `value`, for a local variable declaration in a `set` accessor.

## Remarks

Properties can be marked as `public`, `private`, `protected`, `internal`, `protected internal` or `private protected`. These access modifiers define how users of the class can access the property. The `get` and `set` accessors for the same property may have different access modifiers. For example, the `get` may be `public` to allow read-only access from outside the type, and the `set` may be `private` or `protected`. For more information, see [Access Modifiers](#).

A property may be declared as a static property by using the `static` keyword. This makes the property available to callers at any time, even if no instance of the class exists. For more information, see [Static Classes and Static Class Members](#).

A property may be marked as a virtual property by using the [virtual](#) keyword. This enables derived classes to override the property behavior by using the [override](#) keyword. For more information about these options, see [Inheritance](#).

A property overriding a virtual property can also be [sealed](#), specifying that for derived classes it is no longer virtual. Lastly, a property can be declared [abstract](#). This means that there is no implementation in the class, and derived classes must write their own implementation. For more information about these options, see [Abstract and Sealed Classes and Class Members](#).

### ⓘ Note

It is an error to use a [virtual](#), [abstract](#), or [override](#) modifier on an accessor of a [static](#) property.

# Example

This example demonstrates instance, static, and read-only properties. It accepts the name of the employee from the keyboard, increments `NumberOfEmployees` by 1, and displays the Employee name and number.

C#

 Copy

```
public class Employee
{
    public static int NumberOfEmployees;
    private static int _counter;
    private string _name;

    // A read-write instance property:
    public string Name
    {
        get => _name;
        set => _name = value;
    }

    // A read-only static property:
    public static int Counter => _counter;

    // A Constructor:
    public Employee() => _counter = ++NumberOfEmployees; // Calculate the employee's number:
}

class TestEmployee
{
    static void Main()
    {
        Employee.NumberOfEmployees = 107;
        Employee e1 = new Employee();
        e1.Name = "Claude Vige";

        System.Console.WriteLine("Employee number: {0}", Employee.Counter);
        System.Console.WriteLine("Employee name: {0}", e1.Name);
    }
}
```

```
    }  
}  
/* Output:  
   Employee number: 108  
   Employee name: Claude Vige  
*/
```

## Example

This example demonstrates how to access a property in a base class that is hidden by another property that has the same name in a derived class:

C#

 Copy

```
public class Employee  
{  
    private string _name;  
    public string Name  
    {  
        get => _name;  
        set => _name = value;  
    }  
}  
  
public class Manager : Employee  
{  
    private string _name;  
  
    // Notice the use of the new modifier:  
    public new string Name  
    {  
        get => _name;  
        set => _name = value + ", Manager";  
    }  
}
```

```
class TestHiding
{
    static void Main()
    {
        Manager m1 = new Manager();

        // Derived class property.
        m1.Name = "John";

        // Base class property.
        ((Employee)m1).Name = "Mary";

        System.Console.WriteLine("Name in the derived class is: {0}", m1.Name);
        System.Console.WriteLine("Name in the base class is: {0}", ((Employee)m1).Name);
    }
}
/* Output:
    Name in the derived class is: John, Manager
    Name in the base class is: Mary
*/
```

The following are important points in the previous example:

- The property `Name` in the derived class hides the property `Name` in the base class. In such a case, the `new` modifier is used in the declaration of the property in the derived class:


C#

 Copy

```
public new string Name
```

- The cast `(Employee)` is used to access the hidden property in the base class:

C#

 Copy



```
((Employee)m1).Name = "Mary";
```

For more information about hiding members, see the [new Modifier](#).

## Example

In this example, two classes, `Cube` and `Square`, implement an abstract class, `Shape`, and override its abstract `Area` property. Note the use of the [override](#) modifier on the properties. The program accepts the side as an input and calculates the areas for the square and cube. It also accepts the area as an input and calculates the corresponding side for the square and cube.

C#

 Copy

```
abstract class Shape
{
    public abstract double Area
    {
        get;
        set;
    }
}

class Square : Shape
{
    public double side;

    //constructor
    public Square(double s) => side = s;

    public override double Area
    {
        get => side * side;
        set => side = System.Math.Sqrt(value);
    }
}
```

```
class Cube : Shape
{
    public double side;

    //constructor
    public Cube(double s) => side = s;

    public override double Area
    {
        get => 6 * side * side;
        set => side = System.Math.Sqrt(value / 6);
    }
}

class TestShapes
{
    static void Main()
    {
        // Input the side:
        System.Console.Write("Enter the side: ");
        double side = double.Parse(System.Console.ReadLine());

        // Compute the areas:
        Square s = new Square(side);
        Cube c = new Cube(side);

        // Display the results:
        System.Console.WriteLine("Area of the square = {0:F2}", s.Area);
        System.Console.WriteLine("Area of the cube = {0:F2}", c.Area);
        System.Console.WriteLine();

        // Input the area:
        System.Console.Write("Enter the area: ");
        double area = double.Parse(System.Console.ReadLine());

        // Compute the sides:
        s.Area = area;
```

```
c.Area = area;

// Display the results:
System.Console.WriteLine("Side of the square = {0:F2}", s.side);
System.Console.WriteLine("Side of the cube = {0:F2}", c.side);
}
}
/* Example Output:
Enter the side: 4
Area of the square = 16.00
Area of the cube = 96.00

Enter the area: 24
Side of the square = 4.90
Side of the cube = 2.00
*/
```

## See also

- [C# Programming Guide](#)
- [Properties](#)
- [Interface Properties](#)
- [Auto-Implemented Properties](#)

---

Is this page helpful?

 Yes  No

---