**Lesson 10: Properties**

This lesson teaches C# Properties. Our objectives are as follows:

* Understand What Properties Are For.
* Implement a Property.
* Create a Read-Only Property.
* Create a Write-Only Property.
* Create an auto-implemented property.

**Overview of Properties**

Properties provide the opportunity to protect a field in a class by reading and writing to it through the property. In other languages, this is often accomplished by programs implementing specialized getter and setter methods. C# properties enable this type of protection while also letting you access the property just like it was a field.

Another benefit of properties over fields is that you can change their internal implementation over time. With a public field, the underlying data type must always be the same because calling code depends on the field being the same. However, with a property, you  can change the implementation. For example, if a customer has an ID that is originally stored as an int, you might have a requirements change that made you perform a validation to ensure that calling code could never set the ID to a negative value. If it was a field, you would never be able to do this, but a property allows you to make such a change without breaking code. Now, lets see how to use properties.

**Traditional Encapsultation Without Properties**

Languages that don't have properties will use methods (functions or procedures) for encapsultation. The idea is to manage the values inside of the object, state, avoiding corruption and misuse by calling code. Listing 10-1 demonstrates how this traditional method works, encapsulating *Customer* information via accessor methods.

**Listing 10-1. An Example of Traditional Class Field Access**

using System;

public class Customer

{

private int m\_id = -1;

public int GetID()

{

return m\_id;

}

public void SetID(int id)

{

m\_id = id;

}

private string m\_name = string.Empty;

public string GetName()

{

return m\_name;

}

public void SetName(string name)

{

m\_name = name;

}

}

public class CustomerManagerWithAccessorMethods

{

public static void Main()

{

Customer cust = new Customer();

cust.SetID(1);

cust.SetName("Amelio Rosales");

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.GetID(),

cust.GetName());

Console.ReadKey();

}

}

Listing 10-1 shows the traditional method of accessing class fields. The *Customer* class has four properties, two for each private field that the class encapsulates: *m\_id* and *m\_name*. As you can see, *SetID* and *SetName* assign a new values and *GetID* and *GetName* return values.

Observe how *Main* calls the *SetXxx* methods, which sets *m\_id* to *1* and *m\_name* to "Amelio Rosales" in the *Customer* instance, *cust*.  The call to *Console.WriteLine* demonstrates how to read *m\_id* and *m\_name* from *cust*, via *GetID* and *GetName* method calls, respectively.

This is such a common pattern, that C# has embraced it in the form of a language feature called properties, which you'll see in the next section.

**Encapsulating Type State with Properties**

The practice of accessing field data via methods was good because it supported the object-oriented concept of encapsulation. For example, if the type of *m\_id* or *m\_name* changed from an *int* type to *byte*, calling code would still work. Now the same thing can be accomplished in a much smoother fashion with properties, as shown in Listing 10-2.

**Listing 10-2. Accessing Class Fields With Properties**

using System;

public class Customer

{

private int m\_id = -1;

public int ID

{

get

{

return m\_id;

}

set

{

m\_id = value;

}

}

private string m\_name = string.Empty;

public string Name

{

get

{

return m\_name;

}

set

{

m\_name = value;

}

}

}

public class CustomerManagerWithProperties

{

public static void Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

Listing 10-2 shows how to create and use a property. The *Customer* class has the *ID* and *Name* property implementations. There are also private fields named *m\_id* and *m\_name;* which *ID* and *Name*, respectively, encapsulate. Each property has two accessors, *get* and *set*. The accessor returns the value of a field. The *set* accessor sets the value of a field with the contents of *value*, which is the value being assigned by calling code. The *value* shown in the accessor is a C# reserved word.

When setting a property, just assign a value to the property as if it were a field. The *CustomerManagerWithProperties* class uses the *ID* and *Name* properties in the *Customer* class. The first line of *Main* instantiates a *Customer* object named *cust*. Next the value of the *m\_id* and *m\_name* fields of *cust* are set by using the *ID* and *Name* properties.

To read from a property, use the property as if it were a field. *Console.WriteLine* prints the value of the *m\_id* and *m\_name* fields of *cust*. It does this by calling the *ID* and *Name* properties of *cust*.

This was a read/write property, but you can also create read-only properties, which you'll learn about next.

**Creating Read-Only Properties**

Properties can be made read-only. This is accomplished by having only a *get* accessor in the property implementation. Listing 10-3 demonstrates how you can create a read-only property.

**Listing 10-3. Read-Only Properties**

using System;

public class Customer

{

private int m\_id = -1;

private string m\_name = string.Empty;

public Customer(int id, string name)

{

m\_id = id;

m\_name = name;

}

public int ID

{

get

{

return m\_id;

}

}

public string Name

{

get

{

return m\_name;

}

}

}

public class ReadOnlyCustomerManager

{

public static void Main()

{

Customer cust = new Customer(1, "Amelio Rosales");

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

The *Customer* class in Listing 10-3 has two read-only properties, *ID* and *Name*. You can tell that each property is read-only because they only have *get* accessors. At some time, values for the *m\_id* and *m\_name* must be assigned, which is the role of the constructor in this example.

The *Main* method of the *ReadOnlyCustomerManager* class instantiates a new *Customer* object named *cust*. The instantiation of *cust* uses the constructor of *Customer* class, which takes *int* and *string* type parameters. In this case, the values are *1* and *"Amelio Rosales"*. This initializes the *m\_id* and *m\_name* fields of *cust*.

Since the *ID* and *Name* properties of the *Customer* class are read-only, there is no other way to set the value of the *m\_id* and *m\_name* fields. If you inserted *cust.ID = 7* into the listing, the program would not compile, because *ID* is read-only; the same goes for *Name*. When the *ID* and *Name* properties are used in *Console.WriteLine*, they work fine. This is because these are read operations which only invoke the *get* accessor of the *ID* and *Name* properties.

One question you might have now is "If a property can be read-only, can it also be write-only?" The answer is yes, and explained in the next section.

**Creating a Write-Only Property**

You can assign values to, but not read from, a write-only property. A write-only property only has a *set* accessor. Listing 10-4 shows you how to create and use write-only properties.

**Listing 10-4. Write-Only Properties**

using System;

public class Customer

{

private int m\_id = -1;

public int ID

{

set

{

m\_id = value;

}

}

private string m\_name = string.Empty;

public string Name

{

set

{

m\_name = value;

}

}

public void DisplayCustomerData()

{

Console.WriteLine("ID: {0}, Name: {1}", m\_id, m\_name);

}

}

public class WriteOnlyCustomerManager

{

public static void Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

cust.DisplayCustomerData();

Console.ReadKey();

}

}

This time, the *get* accessor is removed from the *ID* and *Name* properties of the *Customer* class, shown in Listing 10-1. The *set* accessors have been added, assigning *value* to the backing store fields, *m\_id* and *m\_name*.

The *Main* method of the *WriteOnlyCustomerManager* class instantiates the *Customer* class with a default constructor. Then it uses the *ID* and *Name* properties of *cust* to set the *m\_id* and *m\_name* fields of *cust* to *1* and *"Amelio Rosales"*, respectively. This invokes the *set* accessor of *ID* and *Name* properties from the *cust* instance.

When you have a lot of properties in a class or struct, there can also be a lot of code associated with those properties. In the next section, you'll see how to write properties with less code.

**Creating Auto-Implemented Properties**

The patterns you see here, where a property encapsulates a property with *get* and *set* accessors, without any other logic is common. It is more code than we should have to write for such a common scenario. That's why C# 3.0 introduced a new syntax for a property, called an *auto-implemented property*, which allows you to create properties without *get* and *set* accessor implementations. Listing 10-5 shows how to add auto-implemented properties to a class.

**Listing 10-5. Auto-Impemented Properties**

using System;

public class Customer

{

public int ID { get; set; }

public string Name { get; set; }

}

public class AutoImplementedCustomerManager

{

static void Main()

{

Customer cust = new Customer();

cust.ID = 1;

cust.Name = "Amelio Rosales";

Console.WriteLine(

"ID: {0}, Name: {1}",

cust.ID,

cust.Name);

Console.ReadKey();

}

}

Notice how the *get* and *set* accessors in Listing 10-5 do not have implementations. In an auto-implemented property, the C# compiler creates the backing store field behind the scenes, giving the same logic that exists with traditional properties, but saving you from having to use all of the syntax of the traditional property. As you can see in the *Main* method, the usage of an auto-implemented property is exactly the same as traditional properties, which you learned about in previous sections.

**Summary**

You now know what properties are for and how they're used. Traditional techniques of encapsulation have relied on separate methods. Properties allow you to access objects state with field-like syntax. Properties can be made read-only or write-only. You also learned how to write properties with less code by using auto-implemented properties.

This tutorial shows how properties are an integral part of the C# programming language. It demonstrates how properties are declared and used.

## Sample Files

See [Properties Sample](http://msdn.microsoft.com/en-us/library/d50f1h2y(v=vs.71).aspx) to download and build the sample files discussed in this tutorial.

## Further Reading

* [Properties](http://msdn.microsoft.com/en-us/library/x9fsa0sw(v=vs.71).aspx)
* [Comparison Between Properties and Indexers](http://msdn.microsoft.com/en-us/library/4bsztef7(v=vs.71).aspx)
* [10.6 Properties](http://msdn.microsoft.com/en-us/library/aa664450(v=vs.71).aspx)

## Tutorial

This tutorial includes two examples. The first example shows how to declare and use read/write properties. The second example demonstrates abstract properties and shows how to override these properties in subclasses.

#### Example 1

This sample shows a Person class that has two properties: Name (string) and Age (int). Both properties are read/write.

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode0');" \o "Copy to clipboard.)

// person.cs

using System;

class Person

{

private string myName ="N/A";

private int myAge = 0;

// Declare a Name property of type string:

public string Name

{

get

{

return myName;

}

set

{

myName = value;

}

}

// Declare an Age property of type int:

public int Age

{

get

{

return myAge;

}

set

{

myAge = value;

}

}

public override string ToString()

{

return "Name = " + Name + ", Age = " + Age;

}

public static void Main()

{

Console.WriteLine("Simple Properties");

// Create a new Person object:

Person person = new Person();

// Print out the name and the age associated with the person:

Console.WriteLine("Person details - {0}", person);

// Set some values on the person object:

person.Name = "Joe";

person.Age = 99;

Console.WriteLine("Person details - {0}", person);

// Increment the Age property:

person.Age += 1;

Console.WriteLine("Person details - {0}", person);

}

}

#### Output

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode1');" \o "Copy to clipboard.)

Simple Properties

Person details - Name = N/A, Age = 0

Person details - Name = Joe, Age = 99

Person details - Name = Joe, Age = 100

#### Code Discussion

* Notice the way that the properties are declared, for example, consider the Name property:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode2');" \o "Copy to clipboard.)

public string Name

{

get

{

return myName;

}

set

{

myName = value;

}

}

The Set and Get methods of a property are contained inside the property declaration. You can control whether a property is read/write, read-only, or write-only by controlling whether a Get or Set method is included.

* Once the properties are declared, they can be used as if they were fields of the class. This allows for a very natural syntax when both getting and setting the value of a property, as in the following statements:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode3');" \o "Copy to clipboard.)

person.Name = "Joe";

person.Age = 99;

* Note that in a property Set method a special value variable is available. This variable contains the value that the user specified, for example:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode4');" \o "Copy to clipboard.)

myName = value;

* Notice the clean syntax for incrementing the Age property on a Person object:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode5');" \o "Copy to clipboard.)

person.Age += 1;

If separate Set and Get methods were used to model properties, the equivalent code might look like this:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode6');" \o "Copy to clipboard.)

person.SetAge(person.GetAge() + 1);

* The ToString method is overridden in this example:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode7');" \o "Copy to clipboard.)

public override string ToString()

{

return "Name = " + Name + ", Age = " + Age;

}

Notice that ToString is not explicitly used in the program. It is invoked by default by the WriteLine calls.

#### Example 2

The following example shows how to define abstract properties. An abstract property declaration does not provide an implementation of the property accessors. The example demonstrates how to override these properties in subclasses.

This sample consists of three files. In the [Properties Sample](http://msdn.microsoft.com/en-us/library/d50f1h2y(v=vs.71).aspx), these files are compiled into a single compilation but in this tutorial, each file is compiled individually and its resulting assembly referenced by the next compilation:

* abstractshape.cs: The Shape class that contains an abstract Area property.
* shapes.cs: The subclasses of the Shape class.
* shapetest.cs: A test program to display the areas of some Shape-derived objects.

To compile the example, use the command line:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode8');" \o "Copy to clipboard.)

csc abstractshape.cs shapes.cs shapetest.cs

This will create the executable file shapetest.exe.

**File 1 - abstractshape.cs**

This file declares the Shape class that contains the Area property of the type double

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode9');" \o "Copy to clipboard.)

// abstractshape.cs

// compile with: /target:library

// csc /target:library abstractshape.cs

using System;

public abstract class Shape

{

private string myId;

public Shape(string s)

{

Id = s; // calling the set accessor of the Id property

}

public string Id

{

get

{

return myId;

}

set

{

myId = value;

}

}

// Area is a read-only property - only a get accessor is needed:

public abstract double Area

{

get;

}

public override string ToString()

{

return Id + " Area = " + string.Format("{0:F2}",Area);

}

}

#### Code Discussion

* Modifiers on the property are placed on the property declaration itself, for example:

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode10');" \o "Copy to clipboard.)

public abstract double Area

* When declaring an abstract property (such as Area in this example), you simply indicate what property accessors are available, but do not implement them. In this example, only a Get accessor is available, so the property is read-only.

**File 2 - shapes.cs**

The following code shows three subclasses of Shape and how they override the Area property to provide their own implementation.

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode11');" \o "Copy to clipboard.)

// shapes.cs

// compile with: /target:library /reference:abstractshape.dll

public class Square : Shape

{

private int mySide;

public Square(int side, string id) : base(id)

{

mySide = side;

}

public override double Area

{

get

{

// Given the side, return the area of a square:

return mySide \* mySide;

}

}

}

public class Circle : Shape

{

private int myRadius;

public Circle(int radius, string id) : base(id)

{

myRadius = radius;

}

public override double Area

{

get

{

// Given the radius, return the area of a circle:

return myRadius \* myRadius \* System.Math.PI;

}

}

}

public class Rectangle : Shape

{

private int myWidth;

private int myHeight;

public Rectangle(int width, int height, string id) : base(id)

{

myWidth = width;

myHeight = height;

}

public override double Area

{

get

{

// Given the width and height, return the area of a rectangle:

return myWidth \* myHeight;

}

}

}

**File 3 - shapetest.cs**

The following code shows a test program that creates a number of Shape-derived objects and prints out their areas.

[Copy](javascript:CodeSnippet_CopyCode('CodeSnippetContainerCode12');" \o "Copy to clipboard.)

// shapetest.cs

// compile with: /reference:abstractshape.dll;shapes.dll

public class TestClass

{

public static void Main()

{

Shape[] shapes =

{

new Square(5, "Square #1"),

new Circle(3, "Circle #1"),

new Rectangle( 4, 5, "Rectangle #1")

};

System.Console.WriteLine("Shapes Collection");

foreach(Shape s in shapes)

{

System.Console.WriteLine(s);

}

}

}

In C#, properties are nothing but natural extension of data fields. They are usually known as ‘smart fields’ in C# community. We know that data encapsulation and hiding are the two fundamental characteristics of any object oriented programming language. In C#, data encapsulation is possible through either classes or structures. By using various access modifiers like private, public, protected, internal etc it is possible to control the accessibility of the class members.

Usually inside a class, we declare a data field as private and will provide a set of public SET and GET methods to access the data fields. This is a good programming practice, since the data fields are not directly accessible out side the class. We must use the set/get methods to access the data fields.

An example, which uses a set of set/get methods, is shown below.

//SET/GET methods  
//Author: rajeshvs@msn.com  
using System;  
class MyClass  
{  
            private int x;  
            public void SetX(int i)  
            {  
                        x = i;  
            }  
            public int GetX()  
            {  
                        return x;  
            }  
}  
class MyClient  
{  
            public static void Main()  
            {  
                        MyClass mc = new MyClass();  
                        mc.SetX(10);  
                        int xVal = mc.GetX();  
                        Console.WriteLine(xVal);//Displays 10  
            }  
}

But C# provides a built in mechanism called properties to do the above. In C#, properties are defined using the property declaration syntax. The general form of declaring a property is as follows.

<acces\_modifier> <return\_type> <property\_name>  
{  
           get  
           {  
           }  
           set  
           {  
           }  
}

Where <access\_modifier> can be private, public, protected or internal. The <return\_type> can be any valid C# type. Note that the first part of the syntax looks quite similar to a field declaration and second part consists of a get accessor and a set accessor.

For example the above program can be modifies with a property X as follows.

class MyClass  
{  
           private int x;  
           public int X  
           {  
                      get  
                      {  
                                  return x;  
                      }  
                      set  
                      {  
                                  x = value;  
                      }  
           }  
}

The object of the class MyClass can access the property X as follows.

MyClass mc = new MyClass();  
mc.X = 10; // calls set accessor of the property X, and pass 10 as value of the standard field   
//‘value’. This is used for setting value for the data member x.  
Console.WriteLine(mc.X);// displays 10. Calls the get accessor of the property X.

The complete program is shown below.

//C#: Property  
//Author: rajeshvs@msn.com  
using System;  
class MyClass  
{  
            private int x;  
            public int X  
            {  
                        get  
                        {  
                                    return x;  
                        }  
                        set  
                        {  
                                    x = value;  
                        }  
            }  
}  
class MyClient  
{  
            public static void Main()  
            {  
                        MyClass mc = new MyClass();  
                        mc.X = 10;  
                        int xVal = mc.X;  
                        Console.WriteLine(xVal);//Displays 10  
            }  
}

Remember that a property should have at least one accessor, either set or get. The set accessor has a free variable available in it called value, which gets created automatically by the compiler. We can’t declare any variable with the name value inside the set accessor.

We can do very complicated calculations inside the set or get accessor. Even they can throw exceptions.

Since normal data fields and properties are stored in the same memory space, in C#, it is not possible to declare a field and property with the same name.

**Static Properties**

C# also supports static properties, which belongs to the class rather than to the objects of the class. All the rules applicable to a static member are applicable to static properties also.

The following program shows a class with a static property.

//C# : static Property  
//Author: rajeshvs@msn.com  
using System;  
class MyClass  
{  
            private  static int x;  
            public static int X  
            {  
                        get  
                        {  
                                    return x;  
                        }  
                        set  
                        {  
                                    x = value;  
                        }  
            }  
}  
class MyClient  
{  
            public static void Main()  
            {  
                        MyClass.X = 10;  
                        int xVal = MyClass.X;  
                                Console.WriteLine(xVal);//Displays 10  
            }  
}

Remember that set/get accessor of static property can access only other static members of the class. Also static properties are invoking by using the class name.

**Properties & Inheritance**

The properties of a Base class can be inherited to a Derived class.

//C# : Property : Inheritance  
//Author: rajeshvs@msn.com  
using System;  
class Base  
{  
            public int X  
            {  
                        get  
                        {  
                                    Console.Write("Base GET");  
                                    return 10;  
                        }  
                        set  
                        {  
                                    Console.Write("Base SET");  
                        }  
            }  
}  
class Derived : Base  
{  
              
}  
class MyClient  
{  
            public static void Main()  
            {  
                        Derived d1 = new Derived();  
                        d1.X = 10;  
                        Console.WriteLine(d1.X);//Displays 'Base SET Base GET 10'  
            }  
}

The above program is very straightforward. The inheritance of properties is just like inheritance any other member.

**Properties & Polymorphism**

A Base class property can be polymorphicaly overridden in a Derived class. But remember that the modifiers like virtual, override etc are using at property level, not at accessor level.

//C# : Property : Polymorphism  
//Author: rajeshvs@msn.com  
using System;  
class Base  
{  
            public virtual int X  
                {  
                                get  
                        {  
                                    Console.Write("Base GET");  
                                    return 10;  
                        }  
                        set  
                        {  
                                    Console.Write("Base SET");  
                        }  
            }  
}  
class Derived : Base  
{  
            public override int X  
            {  
                        get  
                        {  
                                    Console.Write("Derived GET");  
                                    return 10;  
                        }  
                        set  
                        {  
                                    Console.Write("Derived SET");  
                        }  
            }            
}  
class MyClient  
{  
            public static void Main()  
            {  
                        Base b1 = new Derived();  
                        b1.X = 10;  
                        Console.WriteLine(b1.X);//Displays 'Derived SET Derived GET 10'  
            }  
}

**Abstract Properties**

A property inside a class can be declared as abstract by using the keyword abstract. Remember that an abstract property in a class carries no code at all. The get/set accessors are simply represented with a semicolon. In the derived class we must implement both set and get assessors.

If the abstract class contains only set accessor, we can implement only set in the derived class.

The following program shows an abstract property in action.

//C# : Property : Abstract  
//Author: rajeshvs@msn.com  
using System;  
abstract class Abstract  
{  
            public abstract int X  
            {  
                        get;  
                        set;  
            }  
}  
class Concrete : Abstract  
{  
            public override int X  
            {  
                        get  
                        {  
                                    Console.Write(" GET");  
                                    return 10;  
                        }  
                        set  
                        {  
                                    Console.Write(" SET");  
                        }  
            }            
}  
class MyClient  
{  
            public static void Main()  
            {  
                        Concrete c1 = new Concrete();  
                        c1.X = 10;  
                        Console.WriteLine(c1.X);//Displays 'SET GET 10'  
            }  
}

The properties are an important features added in language level inside C#. They are very useful in GUI programming. Remember that the compiler actually generates the appropriate getter and setter methods when it parses the C# property syntax.