**The C# Station Tutorial**

**by**[**Joe Mayo**](http://twitter.com/#!/JoeMayo)**http://www.csharp-station.com/Tutorial/CSharp/**

## 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 Encapsulation Without Properties

Languages that don't have properties will use methods (functions or procedures) for encapsulation. 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 methods, 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 andset. The get 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 IDand *Name* properties in the Customer class. The first line of Main instantiates a *Customer* object named cust. Next the value of them\_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 ofcust. 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 custuses 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 setaccessors 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 theID 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 setaccessor 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-Implemented 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.