CHAPTER 8
Inside Methods

In Chapter 7, you saw that classes consist of fields and methods. Fields hold the state of the object, and methods define the object's behavior.

In this chapter, you'll explore how methods work in more detail. You've already seen how to create methods, and in this chapter, you'll learn about method *overloading*, a technique that allows you to create more than one method with the same name. This enables your clients to invoke the method with different parameter types.

This chapter also introduces *properties*. To clients of your class, properties look like member variables, but properties are implemented as methods. This allows you to maintain good data-hiding, while providing your clients with convenient access to the state of your class.

Chapter 7 described the difference between value types (such as `int` and `long`) and reference types. The most common value types are the "built-in" or "primitive" types (as well as structs), while the most common reference types are the user-defined types. This chapter explores the implications of passing value types to methods and shows how you can pass value types *by reference*, allowing the called method to act on the original object in the calling method.

Overloading Methods

Often you'll want to have more than one method with the same name. The most common example of this is to have more than one constructor with the same name, which allows you to create the object with different types of parameters, or a different number of parameters. For example, if you were creating a `Time` object, you might have circumstances where you want to create the `Time` object by passing in the date, hours, minutes, and seconds. Other times, you might want to create a `Time` object by passing in an existing `Time` object. Still other times, you might want to pass in just a date, without hours and minutes. Overloading the constructor allows you to provide these various options.

Chapter 7 explained that your constructor is automatically invoked when your object is created. Let's return to the `Time` class created in that chapter, to the client who could create a `Time` object by passing in a `DateTime` object to the constructor.

The `DateTime` object is an object that's built into the `System` library, with many of the same data members as your custom `Time` class. In short, having `DateTime` means you probably won't ever create your own `Time` class, but we're using our custom `Time` class as an example of many of the issues that arise in creating classes.

It would be convenient also to allow the client to create a new `Time` object by passing in year, month, date, hour, minute, and second values. Some clients might prefer one or the other constructor; you can provide both, and the client can decide which better fits the situation.

In order to overload your constructor, you must make sure that each constructor has a unique *signature*. The signature of a method is composed of its name and its parameter list. Two methods differ in their signatures if they have different names or different parameter lists. Parameter lists can differ by having different numbers or types of parameters. The following four lines of code show how you might distinguish methods by signature:

```
void MyMethod(int p1);
void MyMethod(int p1, int p2);    // different number
void MyMethod(int p1, string s1); // different types
void SomeMethod(int p1);          // different name
```

The first three methods are all overloads of the `MyMethod( )` method. The first differs from the second and third in the number of parameters. The second closely resembles the third version, but the second parameter in each is a different type. In the second method, the second parameter (`p2`) is an integer; in the third method, the second parameter (`s1`) is a string. These changes to the number or type of parameters are sufficient changes in the signature to allow the compiler to distinguish the methods.

The fourth method differs from the other three methods by having a different name. This is not method overloading, just different methods, but it illustrates that two methods can have the same number and type of parameters if they have different names. Thus, the fourth method and the first have the same parameter list, but their names are different.

A class can have any number of methods, as long as each one's signature differs from that of all the others. Example 8-1 illustrates a `Time` class with two constructors: one that takes a `DateTime` object and one that takes six integers.

Example 8-1. Overloading a method

```
using System;

namespace MethodOverloading
{
   public class Time
   {
      // private member variables
      private int Year;
      private int Month;
      private int Date;
      private int Hour;
      private int Minute;
      private int Second;

      // public accessor methods
      public void DisplayCurrentTime( )
      {
         System.Console.WriteLine( "{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second );
      }

      // constructors
      public Time( System.DateTime dt )
      {
         Year = dt.Year;
         Month = dt.Month;
         Date = dt.Day;
         Hour = dt.Hour;
         Minute = dt.Minute;
         Second = dt.Second;
      }

      public Time( int Year, int Month, int Date,
      int Hour, int Minute, int Second )
      {
         this.Year = Year;
         this.Month = Month;
         this.Date = Date;
         this.Hour = Hour;
         this.Minute = Minute;
         this.Second = Second;
      }
   }

   class Tester
   {
      public void Run( )
      {
         System.DateTime currentTime = System.DateTime.Now;

         Time time1 = new Time( currentTime );
         time1.DisplayCurrentTime( );
```

Example 8-1. Overloading a method (continued)

```
        Time time2 = new Time( 2000, 11, 18, 11, 03, 30 );
        time2.DisplayCurrentTime( );
    }

    static void Main( )
    {
        Tester t = new Tester( );
        t.Run( );
    }
  }
}
```

The output looks like this:

```
7/10/2008 16:17:32
11/18/2000 11:3:30
```

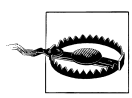
Note that the minutes in the second output show as 3 rather than 03. You can fix this by formatting the output string, left as an exercise for the user.

If a function's signature consisted only of the function name, the compiler would not know which constructors to call when constructing the new `Time` objects, `time1` and `time2`. However, because the signature includes the parameters and their types, the compiler is able to match the constructor call for `time1` with the constructor whose signature requires a `DateTime` object:

```
System.DateTime currentTime = System.DateTime.Now;
Time time1 = new Time(currentTime);
public Time(System.DateTime dt)
```

Likewise, the compiler is able to associate the `time2` constructor call with the constructor whose signature specifies six integer arguments.

```
Time time2 = new Time(2000,11,18,11,03,30);
public Time(int Year, int Month, int Date, int Hour, int Minute, int Second)
```



When you overload a method, you must change the signature (the name, number, or type of the parameters). You are free, as well, to change the return type, but this is optional. Changing only the return type does not overload the method, and creating two methods with the same signature but differing return types generates a compile error.

Encapsulating Data with Properties

It is generally desirable to designate the member variables of a class as `private`. This means that only member methods of that class can access their value. When you

prevent methods outside the class from directly accessing member variables, you're enforcing *data hiding*, which is part of the encapsulation of a class.

Object-oriented programmers are told that member variables should be private. That's fine, but how do you provide access to this data to your clients? The answer for C# programmers is *properties*. Properties allow clients to access class state as if they were accessing member fields directly, while actually implementing that access through a class method.

This solution is ideal. The client wants direct access to the state of the object. The class designer, however, wants to hide the internal state of the class in class fields and provide indirect access through a method. The property provides both the illusion of direct access for the client, and the reality of indirect access for the class developer.

By decoupling the class state from the method that accesses that state, the designer is free to change the internal state of the object as needed. When the `Time` class is first created, the `Hour` value might be stored as a member variable. When the class is redesigned, the `Hour` value might be computed or retrieved from a database. If the client had direct access to the original `Hour` member variable, changing how that value is resolved would break the client. By decoupling and forcing the client to go through a property, the `Time` class can change how it manages its internal state without breaking client code.

In short, properties provide the data hiding required by good object-oriented design. Example 8-2 creates a property called `Hour`, which is then discussed in the paragraphs that follow.

Example 8-2. Properties

```
using System;

namespace Properties
{
   public class Time
   {

      // private member variables
      private int year;
      private int month;
      private int date;
      private int hour;
      private int minute;
      private int second;

      // create a property
      public int Hour
      {
         get
         {
```

Example 8-2. Properties (continued)

```
            return hour;
         }

         set
         {
            hour = value;
         }
      }

      // public accessor methods
      public void DisplayCurrentTime( )
      {
         System.Console.WriteLine(
         "Time: {0}/{1}/{2} {3}:{4}:{5}",
         month, date, year, hour, minute, second );
      }


      // constructors
      public Time( System.DateTime dt )
      {
         year = dt.Year;
         month = dt.Month;
         date = dt.Day;
         hour = dt.Hour;
         minute = dt.Minute;
         second = dt.Second;
      }


   }
   class Tester
   {
      public void Run( )
      {
         System.DateTime currentTime = System.DateTime.Now;
         Time t = new Time( currentTime );
         t.DisplayCurrentTime( );

         // access the hour to a local variable
         int theHour = t.Hour;

         // display it
         System.Console.WriteLine( "Retrieved the hour: {0}",
         theHour );

         // increment it
         theHour++;

         // reassign the incremented value back through
         // the property
         t.Hour = theHour;
```

Example 8-2. Properties (continued)

```
            // display the property
            System.Console.WriteLine( "Updated the hour: {0}", t.Hour );
        }

        [STAThread]
        static void Main( )
        {
            Tester t = new Tester( );
            t.Run( );
        }
    }
}
```

The output should look something like this:

```
Time : 7/10/2008 12:7:43
Retrieved the hour: 12
Updated the hour: 13
```

You create a property by writing the property type and name followed by a pair of braces. Within the braces, you can declare the `get` and `set` accessors. These accessors are very similar to methods, but they are actually part of the property itself. The purpose of these accessors is to provide the client with simple ways to retrieve and change the value of the private member `hour`, as you'll see.

Neither of these accessors has explicit parameters, though the `set` accessor has an *implicit* parameter called `value`, which is used to set the value of the member variable.

By convention, property names are written in Pascal notation (initial uppercase).

In Example 8-2, the declaration of the `Hour` property creates both `get` and `set` accessors:

```
public int Hour
{
     get
     {
        return hour;
     }

 set
     {
        hour = value;
     }
}
```

Each accessor has an *accessor-body*, which does the work of retrieving or setting the property value. The property value might be stored in a database (in which case, the

accessor would do whatever work is needed to interact with the database), or it might just be stored in a private member variable (in this case, hour):

```
private int hour;
```

The get Accessor

The body of the get accessor is similar to a class method that returns an object of the type of the property. In Example 8-2, the accessor for the Hour property is similar to a method that returns an int. It returns the value of the private member variable hour in which the value of the property has been stored:

```
get
{
    return hour;
}
```

In this example, the value of a private int member variable is returned, but you could just as easily retrieve an integer value from a database or compute it on the fly.

Remember, this description is from the perspective of the author of the Time class. To the client (user) of the Time class, Hour is a property, and how the Time class returns its hour is encapsulated within the Time class—the client doesn't know or care.

Whenever you need to retrieve the value (other than to assign to it), the get accessor is invoked. For example, in the following code, the value of the Time object's Hour property is assigned to a local variable.

To the client, the local variable theHour is assigned the value of the Hour property of t (the Time object). To the creator of the Time object, however, the get accessor is called, which, in this case, returns the value of the hour member variable:

```
Time t = new Time(currentTime);
int theHour = t.Hour;
```

The set Accessor

The set accessor sets the value of a property. When you define a set accessor, you must use the value keyword to represent the argument whose value is assigned to the property:

```
set
{
    hour = value;
}
```

Here, again, a private member variable is used to store the value of the property, but the set accessor could write to a database or update other member variables as needed.

When you assign a value to the property, the set accessor is automatically invoked, and the implicit parameter value is set to the value you assign:

```
theHour++;
t.Hour = theHour;
```

The first line increments a local variable named theHour. As far as the client is concerned, that new value is assigned to the Hour property of the local time object t. To the author of the Time class, however, the local variable theHour is passed in to the set accessor as the implicit parameter value and assigned (in this case) to the local member variable hour.

The advantage of this approach is that the client can interact with the properties directly, without sacrificing the data hiding and encapsulation sacrosanct in good object-oriented design.

You can create a read-only property by not implementing the set part of the property. Similarly, you can create a write-only property by not implementing the get part.

Returning Multiple Values

Methods can return only a single value, but this isn't always convenient. Let's return to the Time class. It would be great to create a GetTime( ) method to return the hour, minutes, and seconds. You can't return all three of these as return values, but perhaps you can pass in three parameters, let the GetTime( ) method modify the parameters, and then examine the result in the calling method—in this case, Run( ). Example 8-3 is a first attempt.

Example 8-3. Retrieving multiple values, first attempt

```
using System;

namespace PassByRef
{

   public class Time
   {
      // private member variables
      private int Year;
      private int Month;
      private int Date;
      private int Hour;
      private int Minute;
      private int Second;


      // public accessor methods
      public void DisplayCurrentTime()
      {
```

Example 8-3. Retrieving multiple values, first attempt (continued)

```
            System.Console.WriteLine( "{0}/{1}/{2} {3}:{4}:{5}",
            Month, Date, Year, Hour, Minute, Second );
        }

        public void GetTime(
            int theHour,
            int theMinute,
            int theSecond )
        {
            theHour = Hour;
            theMinute = Minute;
            theSecond = Second;
        }

        // constructor
        public Time( System.DateTime dt )
        {

            Year = dt.Year;
            Month = dt.Month;
            Date = dt.Day;
            Hour = dt.Hour;
            Minute = dt.Minute;
            Second = dt.Second;
        }


    }

    class Tester
    {
        public void Run()
        {
            System.DateTime currentTime = System.DateTime.Now;
            Time t = new Time( currentTime );
            t.DisplayCurrentTime();

            int theHour = 0;
            int theMinute = 0;
            int theSecond = 0;
            t.GetTime( theHour, theMinute, theSecond );
            System.Console.WriteLine( "Current time: {0}:{1}:{2}",
            theHour, theMinute, theSecond );

        }

        static void Main()
        {
            Tester t = new Tester();
            t.Run();
        }
    }
}
```

The output will look something like this:

```
7/1/2008 12:22:19
Current time: 0:0:0
```

Notice that the "Current time" in the output is 0:0:0. Clearly, this first attempt did not work. The problem is with the parameters. You pass in three integer parameters to `GetTime()`, and you modify the parameters in `GetTime()`, but when the values are accessed back in `Run()`, they are unchanged. This is because integers are value types.

Passing Value Types by Reference

As discussed in Chapter 7, C# divides the world of types into value types and reference types. All intrinsic types (such as `int` and `long`) are value types. Instances of classes (objects) are reference types.

When you pass a value type (such as an `int`) into a method, a copy is made. When you make changes to the parameter, you make changes to the copy. Back in the `Run()` method, the original integer variables—`theHour`, `theMinute`, and `theSecond`—are unaffected by the changes made in `GetTime()`.

What you need is a way to pass in the integer parameters by reference so that changes made in the method are made to the original object in the calling method. When you pass an object by reference, the parameter refers to the same object. Thus when you make changes in `GetTime()`, the changes are also made to the original variables in `Run()`.

Please ignore this note as it is advanced, confusing, and put here just to cut down on my email. *Technically*, when you pass a reference type, it is in fact passed by value; but the copy that is made is a copy of a reference, and thus that copy points to the same (unnamed) object on the heap as did the original reference object. That is how you achieve the semantics of "pass by reference" in C# using pass by value.

This is the last time I'll point out this esoteric idea as it is perfectly reasonable to consider the reference to be the object. It takes too long to refer to Fido as a "reference to an unnamed Dog object on the heap"; it is easier to say Fido is a Dog object, and it is reasonable to imagine that Fido is passed by reference, even though (technically) we know better.

Not only is this shorthand reasonable, it is how most professional programmers think and talk about it. In most cases, it comes down to a distinction without a meaningful difference.

This requires two small modifications to the code in Example 8-3. First, change the parameters of the `GetTime()` method to indicate that the parameters are `ref` (reference) parameters:

```
public void GetTime(
   ref int theHour,
```

```
        ref int theMinute,
        ref int theSecond )
    {
        theHour = Hour;
        theMinute = Minute;
        theSecond = Second;
    }
```

Second, modify the call to `GetTime( )` to pass the arguments as references:

```
t.GetTime(ref theHour, ref theMinute, ref theSecond);
```

If you leave out the second step of marking the arguments with the keyword `ref`, the compiler will complain that the argument cannot be converted from an `int` to a `ref int`.

These changes are shown in Example 8-4.

Example 8-4. Passing by reference

```
using System;

namespace PassByRef
{

   public class Time
   {
      // private member variables
      private int Year;
      private int Month;
      private int Date;
      private int Hour;
      private int Minute;
      private int Second;


      // public accessor methods
      public void DisplayCurrentTime()
      {
         System.Console.WriteLine( "{0}/{1}/{2} {3}:{4}:{5}",
         Month, Date, Year, Hour, Minute, Second );
      }

      // takes references to ints
      public void GetTime(
         ref int theHour,
         ref int theMinute,
         ref int theSecond )
      {
         theHour = Hour;
         theMinute = Minute;
         theSecond = Second;
      }
```

Example 8-4. Passing by reference (continued)

```
      // constructor
      public Time( System.DateTime dt )
      {

         Year = dt.Year;
         Month = dt.Month;
         Date = dt.Day;
         Hour = dt.Hour;
         Minute = dt.Minute;
         Second = dt.Second;
      }


   }

   class Tester
   {
      public void Run()
      {
         System.DateTime currentTime = System.DateTime.Now;
         Time t = new Time( currentTime );
         t.DisplayCurrentTime();

         int theHour = 0;
         int theMinute = 0;
         int theSecond = 0;

         // pass the ints by reference
         t.GetTime( ref theHour, ref theMinute, ref theSecond );

         System.Console.WriteLine( "Current time: {0}:{1}:{2}",
         theHour, theMinute, theSecond );

      }

      static void Main()
      {
         Tester t = new Tester();
         t.Run();
      }
   }
}
```

This time, the output looks like this:

```
7/1/2008 12:25:41
Current time: 12:25:41
```

The results now show the correct time.

By declaring these parameters to be `ref` parameters, you instruct the compiler to pass them by reference. Instead of a copy being made, the parameters in `GetTime( )` are references to the corresponding variables (`theHour`, `theMinute`, `theSecond`) that were

created in Run( ). When you change these values in GetTime( ), the change is reflected in Run( ).

Keep in mind that ref parameters are references to the actual original value—it is as if you said, "here, work on this one." Conversely, value parameters are copies—it is as if you said, "here, work on one *just like* this."

out Parameters and Definite Assignment

As noted in Chapter 4, C# imposes *definite assignment*, which requires that all variables be assigned a value before they are used. In Example 8-4, you initialize theHour, theMinute, and theSecond before you pass them as parameters to GetTime( ), yet the initialization merely sets their values to 0 before they are passed to the method:

```
int theHour = 0;
int theMinute = 0;
int theSecond = 0;
t.GetTime( ref theHour, ref theMinute, ref theSecond);
```

It seems silly to initialize these values because you immediately pass them by reference into GetTime( ) where they'll be changed, but if you don't, the following compiler errors are reported:

```
Use of unassigned local variable 'theHour'
Use of unassigned local variable 'theMinute'
Use of unassigned local variable 'theSecond'
```

C# provides the out modifier for situations like this, in which initializing a parameter is only a formality. The out modifier removes the requirement that a reference parameter be initialized. The parameters to GetTime( ), for example, provide no information to the method; they are simply a mechanism for getting information out of it. Thus, by marking all three as out parameters using the out keyword, you eliminate the need to initialize them outside the method.

Within the called method, the out parameters must be assigned a value before the method returns. Here are the altered parameter declarations for GetTime( ):

```
public void GetTime(
   out int theHour,
   out int theMinute,
   out int theSecond )
{
   theHour = Hour;
   theMinute = Minute;
   theSecond = Second;
}
```

Here is the new invocation of the method in Main( ):

```
int theHour;
int theMinute;
int theSecond;
t.GetTime( out theHour, out theMinute, out theSecond);
```

The keyword out implies the same semantics as the keyword ref, except that it also allows you to use the variable without first initializing it in the calling method.

Summary

- Overloading is the act of creating two or more methods with the same name, but that differ in the number and/or type of parameters.
- Properties appear to clients to be members, but appear to the designer of the class to be methods. This allows the designer to modify how the property retrieves its value without breaking the semantics of the client program.
- Properties include get and set accessors that are used to retrieve and modify a member field, respectively. The set accessor has an implicit parameter named value that represents the value to be assigned through the property.
- When you "pass by reference," the called method affects the object referred to in the calling method. When you pass by value, the changes in the called method are not reflected in the calling method. You can pass value types by reference by using either the ref or the out keyword.
- The out parameter eliminates the requirement to initialize a variable before passing it to a method.

Quiz

Question 8-1. What is method overloading and how must the overloaded methods differ?

Question 8-2. What is the signature of a method?

Question 8-3. What are properties?

Question 8-4. How do you create a read-only property?

Question 8-5. How do you retrieve more than one return value from a method?

Question 8-6. Where must you use the keyword ref?

Question 8-7. What is the keyword out used for?

Exercises

Exercise 8-1. Write a program with an overloaded method for doubling the value of the argument. One version of the method should double an `int` value, and the other version should double a `float` value. Call both methods to demonstrate that they work.

Exercise 8-2. Write a program with one method that takes an `int` value, and returns both double and triple that value. You'll need to use reference parameters.

Exercise 8-3. Modify the program from Exercise 8-2 so that you don't need to initialize the variables that will hold the doubled and tripled values before calling the method.