WEEK 1

DAY 6

Classes

As you learned on Day 2, "Understanding C# Programs," classes are critical to an object-oriented language. Classes are also critical to C#. You have seen classes used in every example included in the book so far. Today you

- Revisit the concepts involved in object-oriented programming
- Learn how to declare a class
- Learn how to define a class
- Discover class members
- Create your own data members
- Implement properties in your classes
- Take your first serious look at namespaces

Object-Oriented Programming Revisited

On Day 2, you learned that C# is considered an object-oriented language. You also learned that to take full advantage of C#, you should understand the concepts of object-oriented languages. In the next few sections, you briefly revisit the concepts you learned about in Day 2. You will then begin to see how these concepts are applied to actual C# programs.

Recall from Day 2 the key characteristics that make up an object-oriented language:

- Encapsulation
- Polymorphism
- Inheritance
- Reuse

Encapsulation

Encapsulation is the concept of making classes (or "packages") that contain everything you need. In object-oriented programming, this means that you can create a class that stores all the variables that you need and all the routines to commonly manipulate this data. You can create a `Circle` class that stores information on a circle. This could include storing the location of the circle's center and its radius plus storing routines commonly used with a circle. These routines could include getting the circles area, getting its circumference, changing its center point, changing its radius, and much more.

By encapsulating a circle, you allow the user to be oblivious to how the circle works. You need to know only how to interact with the circle. This provides a shield to the inner workings of the circle, which means that the variables within the class could be changed and it would be invisible to the user. For example, instead of storing the radius of the circle, you could store the diameter. If you have encapsulated the functionality and the data, making this change impacts only your class. Any programs that use your class should not need to change. In today's and tomorrow's lessons, you see programs that work directly with a `Circle` class.

Note

> Encapsulation is often referred to as "black boxing." Black boxing refers to hiding the functionality or the inner workings of a process. For a circle, if you send in the radius, you can get the area. You don't care how it happens, as long as you know you are getting back the correct answer.

Polymorphism

Polymorphism means to have the capability of assuming many forms, which means that the programs can work with what you send them. For example, you have used the `WriteLine()` routine in several of the previous days. You have seen that you can create a parameter field using `{0}`. What values does this field print? As you have seen, it can print a variable regardless of its type or it can print another string. The `WriteLine()` routine takes care of how it gets printed. The routine is polymorphic in that it adapts to most of the types you can send it.

Using a circle as an example, you might want to call a circle object to get its area. You can do this by using three points or by using a single point and the radius. Either way, you expect to get the same results. Additionally, you know that a circle is a shape. As such, a polymophic characteristic of a circle is to be capable of understanding and reacting as a shape.

Inheritance

As you learned in Day 2, inheritance is the most complicated of the object-oriented concepts. Inheritance is when one class (object) is an expansion of another.

In many object-oriented programming books, an animal analogy is used to illustrate inheritance. The analogy starts with the concept of an animal as a living being.

Now consider reptiles, which are everything that an animal is, plus they are cold-blooded. A reptile contains all of the features of an animal, but it also adds its own unique features. Now consider a snake. A snake is a reptile that is long and skinny that has no legs. It has all the characteristics of a reptile, but it also has its own unique characteristics. A snake can be said to inherit the characteristics of a reptile. A reptile can be said to inherit the characteristics of an animal.

On Day 11, "Inheritance," you will see how this same concept is applied to classes and programming.

Reuse

When you create a class, you can reuse it to create lots of objects. By using inheritance and some of the features described previously, you can create routines that can be used repeatedly in many programs and in many ways. By encapsulating functionality, you can create routines that have been tested and proven to work. You won't have to test the details of how the functionality works, only that you are using it correctly. This makes reusing these routines quick and easy.

Objects and Classes

On Day 2, an illustration of a cookie cutter and cookies were used to illustrate classes and objects. Now you are done with cookies and snakes—it is time to jump into some code.

Over the next three days you are going to learn about classes, starting with extremely simple examples and building on them over the next several days.

Defining a Class

To keep things simple, a keyword called `class` is used to define classes. The basic structure of a class is in the following format:

```
class identifier
{
     class-body ;
}
```

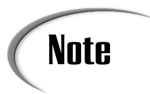
where *`identifier`* is the name given to the class and *`class-body`* is the code that makes up the class.

The name of a class is like any other variable name that can be declared. You want to give a class a meaningful name, something that describes what the class does.

The Microsoft .NET framework has a large number of built-in classes. You have actually been using one since the beginning of this book: the `Console` class. The `Console` class contains several data members and routines. You've already used many of these routines, including `Write` and `WriteLine`. The class name—the *identifier*—of this class is `Console`. The body of the `Console` class contains the code for the `Write` and `WriteLine` routines. By the end of tomorrow's lesson, you will be able to create and name your own classes that have routines similar to the `Console` class.

Class Declarations

After a class is defined, you use it to create objects. A class is just a definition used to create objects. A class by itself does not have the capability of holding information or actually performing routines. Rather, a class is used to declare objects. The object can then be used to hold the data and perform the routines as defined by the class.



The declaration of an object is commonly referred to as *instantiation*. Said differently, an object is an instance of a class.

The format of declaring an object from a class is as follows:

```
class_name object_identifier = new class_name();
```

where *`class_name`* is the name of the class and *`object_identifier`* is the name of the object being declared. For example, if you have a class called `point`, you could create an object called `startingPoint` with the following line of code:

```
point startingPoint = new point();
```

The name of the class is `point`, the name of the object declared is `startingPoint`. Because `startingPoint` is an object, it can contain data and routines if they were defined within the `point` class.

In looking at this declarative line of code, you might wonder what the other items are. Most importantly there is a keyword being used that you have not seen before; `new`.

As its name implies, the `new` keyword is used to create new items. In this case it creates a new point. Because `point` is a class, an object is created. The new keyword indicates that a new instance is to be created. In this case, the new instance is a `point` object.

When declaring an object with a class, you also have to provide parentheses to the class name on the right of the assignment. This enables the class to be constructed into a new object.

If you don't add the construction code, `new classname`, you will have declared a class, but the compiler won't have constructed its internal structure. You need to make sure you assign the `new classname` code to the declared object name to make sure everything is constructed. You will learn more about this initial construction in tomorrow's lesson.

Look at the statement again:

```
point startingPoint = new point();
```

The following breaks down what is happening:

```
point startingPoint
```

The `point` class is used to declare an object called `startingPoint`. This piece of the statement is like what you have seen with other data types, such as integers and decimals.

```
startingPoint =
```

As with variables, you assign the result of the right side of the assignment operator (the equal sign) to the variable on the left. In this case, the variable happens to be an object—which you now know is an object of type `point` called `startingPoint`.

```
new point()
```

This part of the statement does the actual construction of the `point` object. The class name with parentheses is a signal to construct—create—an object of the class type. The

new keyword says to reserve some room in memory of this new object. Remember, a class is only a definition: it doesn't store anything. The object needs to store information, so it needs memory reserved. The new keyword reserves the memory.

Like all statements, this declaration is ended with a semicolon, which signals that the statement is done.

The Members of a Class

Now that you know the overall structure of a class and how to create objects with a class, it is time to look at what can be held in a class. There are two primary types of items that can be contained within the body of a class: data members and function members.

Data members include variables and constants. These include variables of any of the types you learned about on Day 3, "Storing Information with Variables," and any of the more advanced types you will learn about later. These data members can even be other classes.

The other type of element that is part of a class's body is function members. Function members are routines that perform an action. These actions can be as simple as setting a value to something more complex, such as writing a line of text using a variable number of values—as you have seen with Write and WriteLine. Write and WriteLine are member functions of the Console class. In tomorrow's lesson, you will learn how to create and use member functions of your own. For now, it is time to visit data members.

Data Members, aka Fields

NEW TERM Another name for a variable is a *field*. As stated previously, data members within a class are variables that are members of a class.

In the point class referenced earlier, you expect a data member to store the x and y coordinates of the point. These coordinates could be any of a number of data types; however, if these were integers, you define the point class as such:

```
class point
{
   int x;
   int y;
}
```

That's it. This is effectively the code for a very simple point class. There is one other item that you should include for now. This is an access modifier called public. Without adding the word public, you cannot access x or y outside the point class. A variable is accessible only within the block you declare it, unless you indicate otherwise. In this case, the block is the definition of the point class.

Remember, a block is a section of code between two braces {}. The body of a class is a block of code.

The change made to the `point` class is relatively simple. With the `public` accessor added, the class becomes

```
class point
{
   public int x;
   public int y;
}
```

Although the point class contains two integers, you can actually use any data type within this class. For example, you can create a `FullName` class that contains three strings that store the first, middle, and last names. You can create an `Address` class that contains a name class and additional strings to hold the different address pieces. You can create a customer class that contains a `long` value for a customer number, and an address class, a decimal account balance, a Boolean value for active or inactive, and more.

Accessing Data Members

When you have data members declared, you want to get to their values. As you learned, the `public` accessor enables you to get to the data members from outside the class.

You cannot simply access data members from outside the class by their name. For example, if you have a program that declares a `startingPoint` from the `point` class, it would seem as if you should be able to get the point by using `x` and `y`—the names that are in the `point`. What happens if you declare both a `startingPoint` and an `endingPoint` in the same program? If you use `x`, which point is being accessed?

To access a data member, you use both the name of the object and the data member. The member operator, which is a period, separates these. To access the `startingPoint`'s coordinates, you therefore use

```
startingPoint.x
```

and

```
startingPoint.y
```

For the ending point, you use

```
endingPoint.x
```

and

```
endingPoint.y
```

At this time, you have the foundation to try out a program. Listing 6.1 presents the `point` class. This class is used to declare two objects, `starting` and `ending`.

LISTING 6.1 point.cs—Declaring a Class with Data Members

```
 1:  //  point.cs- A class with two data members
 2:  //-----------------------------------------------------------------
 3:
 4:  class point
 5:  {
 6:      public int x;
 7:      public int y;
 8:  }
 9:
10:  class pointApp
11:  {
12:     public static void Main()
13:     {
14:        point starting = new point();
15:        point ending   = new point();
16:
17:        starting.x = 1;
18:        starting.y = 4;
19:        ending.x = 10;
20:        ending.y = 11;
21:
22:        System.Console.WriteLine("Point 1: ({0},{1})",
23:                                 starting.x, starting.y);
24:        System.Console.WriteLine("Point 2: ({0},{1})",
25:                                 ending.x, ending.y);
26:     }
27:  }
```

OUTPUT

```
Point 1: (1,4)
Point 2: (10,11)
```

ANALYSIS A simple class called `point` is declared in lines 4 to 8.This class follows the structure that was presented earlier. In line 4, the `class` keyword is being used, followed by the name of the class, `point`. Lines 5 and 8 contain the braces that enclose the body of the class. Within the body of this class, two integers are declared, `x` and `y`. These are each declared as `public` so that you can use them outside of the class.

Line 10 contains the start of the main portion of your application. It is interesting to note that the main portion of your application is also a class! You will learn more about this later.

Line 12 contains the main routine that you should now be very familiar with. In lines 14 and 15, two objects are created using the `point` class, which follow the same format that

was described earlier. In lines 17 to 20, values are set for each of the data members of the `point` objects. In line 17, the value `1` is assigned to the `x` data member of the starting class. The member operator, the period, separates the member name from the object name. Lines 18, 19, and 20 follow the same format.

Line 22 contains a `WriteLine` routine, which you have also seen before. This one is unique because you print the values stored within the `starting point` object. The values are stored in `starting.x` and `starting.y`, not just `x` and `y`. Line 24 prints the values for the `ending point`.

Using Data Members

Listing 6.1 showed you how to assign a value to a data member as well as how to get its value. What if you want to do something more complex then a simple assignment or a simple display?

The data members of a class are like any other variable type. You can use them in operations, control statements, or anywhere that a regular variable can be accessed. Listing 6.2 expands on the use of the point class. In this example, the calculation is performed to determine the length of a line between two points. If you've forgotten your basic algebraic equation for this, Figure 6.1 illustrates the calculation to be performed.

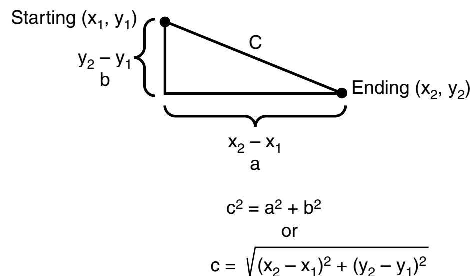


FIGURE 6.1
Calculating line length from two points.

6

LISTING 6.2 point2.cs—Working with Data Members

```
 1:  //  line.cs- Calculate the length of a line.
 2:  //-----------------------------------------------------------------
 3:
 4:  class point
 5:  {
 6:      public int x;
 7:      public int y;
 8:  }
 9:
10:  class lineApp
```

LISTING 6.2 continued

```
11:  {
12:     public static void Main()
13:     {
14:        point starting = new point();
15:        point ending   = new point();
16:        double line;
17:
18:        starting.x = 1;
19:        starting.y = 4;
20:        ending.x = 10;
21:        ending.y = 11;
22:
23:        line = System.Math.Sqrt( (ending.x - starting.x)*(ending.x -
➥starting.x) +
24:                                 (ending.y - starting.y)*(ending.y -
➥starting.y) );
25:
26:        System.Console.WriteLine("Point 1: ({0},{1})",
27:                                  starting.x, starting.y);
28:        System.Console.WriteLine("Point 2: ({0},{1})",
29:                                  ending.x, ending.y);
30:        System.Console.WriteLine("Length of line from Point 1 to Point 2:
➥{0}",
31:                                  line);
32:     }
33:  }
```

OUTPUT

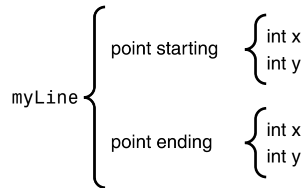
```
Point 1: (1,4)
Point 2: (10,11)
Length of line from Point 1 to Point 2: 11.4017542509914
```

ANALYSIS This listing is very similar to Listing 6.1. The biggest difference is the addition of a data member and some calculations that determine the length of a line. In line 16, you see that the new data member is declared of type `double` and called `line`. This variable will be used to hold the result of the length of the line between the two declared points.

Lines 23 and 24 are actually a single statement. This statement looks more complex than it is. Other than the `System.Math.Sqrt` part, you should be able to follow what the line is doing. `Sqrt` is a routine within the `System.Math` object that calculates the square root of a value. If you compare this formula to Figure 6.2, you will see that it is a match. The end result is the length of the line. The important thing to note is that the data members are being used within this calculation in the same manner that any other variable would be used. The only difference is the naming scheme.

FIGURE 6.2

The `myLine` *class's data members.*



Using Classes as Data Members

It was stated earlier that you can nest one class within another. A class is another type of data. As such, an object declared with a class type—which is just an advanced variable type—can be used in the same places as any other variable. Listing 6.3 presents an example of a line class. This class is composed of two points, `starting` and `ending`.

LISTING 6.3 line2.cs—Nested Classes

```
 1:  //  line2.cs- A class with two data members
 2:  //-------------------------------------------------------------------
 3:
 4:  class point
 5:  {
 6:      public int x;
 7:      public int y;
 8:  }
 9:
10:  class line
11:  {
12:      public point starting = new point();
13:      public point ending = new point();
14:  }
15:
16:  class lineApp
17:  {
18:     public static void Main()
19:     {
20:        line myLine = new line();
21:
22:        myLine.starting.x = 1;
23:        myLine.starting.y = 4;
24:        myLine.ending.x = 10;
25:        myLine.ending.y = 11;
26:
27:        System.Console.WriteLine("Point 1: ({0},{1})",
28:                        myLine.starting.x, myLine.starting.y);
29:        System.Console.WriteLine("Point 2: ({0},{1})",
30:                        myLine.ending.x, myLine.ending.y);
31:     }
32:  }
```

6

Output

```
Point 1: (1,4)
Point 2: (10,11)
```

Analysis Listing 6.3 is very similar to the previous listings. The `point` class that you are coming to know and love is defined in lines 4 to 8. There is nothing different about this from what you have seen before. In lines 10 to 14, however, you see a second class being defined. This class, called `line`, is composed of two variables that are of type `point`, which is a class. These two variables are called `starting` and `ending`. When an object is declared using the `line` class, the `line` class will in turn create two `point` objects.

Continuing with the listing, you see in line 20 that a new line is called. This new line is given the name `myLine`. Line 20 follows the same format you saw earlier for creating an object from a class.

Lines 22 to 25 access the data members of the `line` class and assign them values. It is beginning to look a little more complex; however, looks can be deceiving. If you break this down, you will see that it is relatively straightforward. In line 22, you assign the constant value `1` to the variable `myLine.starting.x`. In other words, you are assigning the value `1` to the `x` member of the `starting` member of `myLine`. Going from the other direction, you can say that you are assigning the value `1` to the `myLine` object's `starting` member's `x` member. It is like a tree. Figure 6.2 illustrates the `myLine` class's members and their names.

Nested types

On Day 3, you learned about the different standard data types that could be used. As you saw in Listing 6.3, an object created with a class can be used in the same places as any other variable created with a data type.

When used by themselves, classes really do nothing—they are only a description. For example, in Listing 6.3, the `point` class in lines 4 to 8, is only a description; nothing is declared and no memory is used. This description defines a type. In this case, the type is the class, or specifically a `point`.

It is possible to nest a type within another class. If `point` is going to be used only within the context of a line, it could be defined within the `line` class. This would enable `point` objects to be used in the `line` class.

The code for the nested `point` type is

```
class line
{
    public class point
    {
```

```
        public int x;
        public int y;
    }

    public point starting = new point();
    public point ending = new point();
}
```

One additional change was made. The `point` class had to be declared as `public` as well. If you don't declare the type as `public`, you get an error. The reason for the error should make sense if you think about it. How can the parts of the `point` or the `point` objects be `public` if the `point` itself isn't `public`?

Static Variables

There are times when you will want a bunch of objects declared with the same class to share a value. For example, you might want to declare a number of line objects that all share the same originating point. If one `line` object changes the originating point, you want all lines to change it.

To share a single data value across all the objects declared by a single class, you add the `static` modifier. Listing 6.4 revisits the `line` class. This time, the same starting point is used for all objects declared with the `line` class.

LISTING 6.4 statline.cs—Using the `static` Modifier with Data Members

```
 1:  //  statline.cs- A class with two data members
 2:  //--------------------------------------------------------------------
 3:
 4:  class point
 5:  {
 6:     public int x;
 7:     public int y;
 8:  }
 9:
10:  class line
11:  {
12:      static public point origin= new point();
13:      public point ending = new point();
14:  }
15:
16:  class lineApp
17:  {
18:     public static void Main()
19:     {
20:        line line1 = new line();
```

LISTING 6.4 continued

```
21:        line line2 = new line();
22:
23:        // set line origin
24:        line.origin.x = 1;
25:        line.origin.y = 2;
26:
27:
28:        // set line1's ending values
29:        line1.ending.x = 3;
30:        line1.ending.y = 4;
31:
32:        // set line2's ending values
33:        line2.ending.x = 7;
34:        line2.ending.y = 8;
35:
36:        // print the values...
37:        System.Console.WriteLine("Line 1 start: ({0},{1})",
38:                                 line.origin.x, line.origin.y);
39:        System.Console.WriteLine("line 1 end:   ({0},{1})",
40:                                 line1.ending.x, line1.ending.y);
41:        System.Console.WriteLine("Line 2 start: ({0},{1})",
42:                                 line.origin.x, line.origin.y);
43:        System.Console.WriteLine("line 2 end:   ({0},{1})\n",
44:                                 line2.ending.x, line2.ending.y);
45:
46:        // change value of line2's starting point
47:        line.origin.x = 939;
48:        line.origin.y = 747;
49:
50:        // and the values again...
51:
52:        System.Console.WriteLine("Line 1 start: ({0},{1})",
53:                                 line.origin.x, line.origin.y);
54:        System.Console.WriteLine("line 1 end:   ({0},{1})",
55:                                 line1.ending.x, line1.ending.y);
56:        System.Console.WriteLine("Line 2 start: ({0},{1})",
57:                                 line.origin.x, line.origin.y);
58:        System.Console.WriteLine("line 2 end:   ({0},{1})",
59:                                 line2.ending.x, line2.ending.y);
60:    }
61: }
```

OUTPUT

```
Line 1 start: (1,2)
line 1 end:   (3,4)
Line 2 start: (1,2)
line 2 end:   (7,8)
```

```
Line 1 start: (939,747)
line 1 end:   (3,4)
Line 2 start: (939,747)
line 2 end:   (7,8)
```

If you try to access a static data member with an object name, such as `line1`, you will get an error. You must use the class name to access a static data member.

ANALYSIS Listing 6.4 is not much different from what you have seen before. The biggest difference is in line 12, where the `origin` point is declared as `static` in addition to being `public`. The `static` keyword makes a big difference in this `line` class. Instead of each object that is created from the `line` class containing an origin point, there is only one origin point that is shared by all instances of `line`.

Line 18 is the beginning of your `Main` routine. Lines 20 and 21 declare two `line` objects called `line1` and `line2`. Lines 28 and 29 set the ending point of `line1`, and lines 33 and 34 set the ending point of `line2`. Going back to lines 24 and 25, you see something different from what you have seen before. Instead of setting the origin point of `line1` or `line2`, these lines set the point for the class name, `line`. This is important. If you try to set the origin on `line1` or `line2`, you will get a compiler error. In other words, the following line of code is an error:

```
line1.origin.x = 1;
```

Because the `origin` object is declared `static`, it is shared across all objects of type `line`. Because neither `line1` nor `line2` own this value, they cannot be used directly to set the value. Rather, you must use the class name. Remember, a variable declared `static` in a class is owned by the class, not the individual objects that are instantiated.

Lines 37 to 44 print the `origin point` and `ending point` for `line1` and `line2`. Again, notice that the class name is used to print the `origin` values, not the object name. Lines 47 and 48 change the `origin`, and the final part of the program prints the values again.

A common use of a `static` data member is as a counter. Each time an object does something, it can increment the counter for all the objects.

The Application Class

If you haven't already noticed, there is a class being used in all your applications that has not been discussed. If you look at line 16 of Listing 6.4, you see the following code:

```
class lineApp
```

You will notice a similar class line in every application you have entered in this book. C# is an object-oriented language. This means everything is an object—even your application. To create an object, you need a class to define it. Listing 6.4's application is `lineApp`. When you execute the program, the `lineApp` class is instantiated and creates a `lineApp` object, which just happens to be your program!

Like what you have learned above, your application class declares data members. In Listing 6.4, the `lineApp` class's data members are two classes: `line1` and `line2`. There is additional functionality in this class as well. In tomorrow's lesson, you will learn that this additional functionality can be included in your classes as well.

Properties

Earlier it was stated that one of the benefits of an object-oriented program is the ability to control the internal representation or access to data. In the examples used so far in today's lesson, everything has been `public`, so access has been freely given to any code that wants to access the data members.

In an object-oriented program, you want to have more control over who can and can't get to data. In general, you won't want code to access data members directly. If you allow code to directly access these data members, you might lock yourself into being unable to change the data types of the values.

C# provides a concept called *properties* to enable you to create object-oriented fields within your classes. Properties use the keywords `get` and `set` to get the values from your variables and set the values in your variables. Listing 6.5 illustrates the use of `get` and `set` with the `point` class that you used earlier.

LISTING 6.5 prop.cs—Using Properties

```
1:  //  prop.cs- Using Properties
2:  //----------------------------------------------------------------
3:
4:  class point
5:  {
6:      int my_X;  // my_X is private
7:      int my_Y;  // my_Y is private
```

LISTING 6.5 continued

```
 8:
 9:      public int x
10:      {
11:         get
12:         {
13:            return my_X;
14:         }
15:         set
16:         {
17:            my_X = value;
18:         }
19:      }
20:      public int y
21:      {
22:         get
23:         {
24:            return my_Y;
25:         }
26:         set
27:         {
28:            my_Y = value;
29:         }
30:      }
31:  }
32:
33:  class MyApp
34:  {
35:     public static void Main()
36:     {
37:        point starting = new point();
38:        point ending   = new point();
39:
40:        starting.x = 1;
41:        starting.y = 4;
42:        ending.x = 10;
43:        ending.y = 11;
44:
45:        System.Console.WriteLine("Point 1: ({0},{1})",
46:                                 starting.x, starting.y);
47:        System.Console.WriteLine("Point 2: ({0},{1})",
48:                                 ending.x, ending.y);
49:     }
50:  }
```

OUTPUT

```
Point 1: (1,4)
Point 2: (10,11)
```

ANALYSIS Listing 6.5 creates properties for both the x and y coordinates of the point class. The point class is defined in lines 4 to 31. Everything on these lines is a part of the point class's definition. In lines 6 and 7, you see that two data members are created. These are called my_X and my_Y. Because these are not declared as public, they cannot be accessed outside the class. They are considered private variables. You will learn more about keeping things private on Day 8, " Advanced Data Storage: Structures, Enumerators, and Arrays."

Lines 9 to 19 and lines 20 to 30 operate exactly the same, except the first set of lines uses the my_X variable and the second set uses the my_Y variable. These sets of lines create the property abilities for the my_X and my_Y variables.

Line 9 looks like just another declaration of a data member. In fact, it is. In this line, you declare a public integer variable called x. Note that there is no semicolon at the end of this line; therefore, the declaration of the member variable is not complete. Rather, it also includes what is in the following code block in lines 10 to 19. Within this block of code you have two commands. Line 11 is a get statement, which is called whenever a program tries to get the value of the data member being declared—in this case, x. For example, if you assign the value of x to a different variable, you get the value of x and set it into the new variable. The set statement in line 15 is called whenever you are setting a value into the x variable. For example, setting x equal to 10 places the value of 10 into x.

When a program gets the value of x, the get property in line 11 is called. This executes the code within the get, which is line 13. Line 13 returns the value of my_X, which is the private variable in the point class.

When a program places a value into x, the set property in line 15 is called. This executes the code within the set, which is line 17. Line 17 sets something called value into the private variable, my_X, in the point class. value is the value being placed into x. (It is great when a name actually describes the contents.) For example, value is 10 in the following statement:

```
x = 10;
```

This statement places the value of 10 into x. The set property within x places this value into my_X.

Looking at the main application in lines 33 to 50, you should see that x is used as it had been used before. There is absolutely no difference to how you use the point class. The difference is that the point class could be changed to store my_X and my_Y differently and it would not impact the program.

Although the code in lines 9 to 30 is relatively simple, it doesn't have to be. You can do any coding and any manipulation you want within the get and set. You don't even have to write to another data member!

Do

DO make sure you understand data members and the class information presented in today's lesson before going to Day 7.

DO use property accessors to access your class's data members in programs you create.

A First Look at Namespaces

As you begin to learn about classes, it is important to know that there is a large number of classes available that do a wide variety of functions. The .NET framework provides a substantial number of base classes that you can use. Additionally, you can obtain third-party classes that you can use.

Day 16, "Using the .NET Base Classes" focuses specifically on using a number of key .NET base classes.

As you continue through this book, you will be exposed to a number of key classes. You've actually used a couple of base classes already. As mentioned earlier, `Console` is a base class. You also learned that `Console` has a number of member routines called `Write` and `WriteLine`. For example, the following writes my name to the console:

```
System.Console.WriteLine("Bradley L. Jones");
```

You now know that `"Bradley L. Jones"` is a literal. You know that `WriteLine` is a routine that is a part of the `Console` class. You even know that `Console` is an object declared from a class. This leaves `System`.

Because of the number of classes, it is important that they be organized. Classes can be grouped together into namespaces. A namespace is a named grouping of classes. The `Console` class is a part of the `System` namespace.

`System.Console.WriteLine` is a fully qualified name. With a fully qualified name, you point directly to where the code is located. C# provides a shortcut method for using classes and methods that doesn't require you to always include the full namespace name. This is accomplished with the `using` keyword.

The `using` keyword enables you to include a namespace in your program. When the namespace is included, the program knows to search the namespace for routines and classes that might be used. The format for including a namespace is

```
using namespace_name
```

where *namespace_name* is the name of the namespace or the name of a nested namespace. For example, to include the `System` namespace, you include the following line of code near the top of your listing:

```
using System;
```

If you include this line of code, you do not need to include the `System` section when calling classes or routines within the namespace. Listing 6.6 calls the `using` statement to include the `System` namespace.

LISTING 6.6 namesp.cs—Using `using` and Namespaces

```
 1:  //  namesp.cs- Namespaces and the using keyword
 2:  //-----------------------------------------------------------------
 3:
 4:  using System;
 5:
 6:  class name
 7:  {
 8:      public string first;
 9:      public string last;
10:  }
11:
12:  class NameApp
13:  {
14:     public static void Main()
15:     {
16:        // Create a name object
17:        name you = new name();
18:
19:        Console.Write("Enter your first name and press enter: ");
20:        you.first = Console.ReadLine();
21:        System.Console.Write("\n{0}, enter your last name and press enter: ",
22:                              you.first);
23:        you.last = System.Console.ReadLine();
24:
25:        Console.WriteLine("\nData has been entered.....");
26:        System.Console.WriteLine("You claim to be {0} {1}",
27:                                  you.first, you.last);
28:     }
29:  }
```

OUTPUT

```
Enter your first name and press enter: Bradley

Bradley, enter your last name and press enter: Jones

Data has been entered.....
You claim to be Bradley Jones
```

The bold text in the output is text that I entered. You can enter any text in its place. I suggest your own name rather than mine!

ANALYSIS Line 4 of Listing 6.6 is the focus point of this program. The `using` keyword includes the `System` namespace; when you use functions from the `Console` class, you don't have to fully qualify their names. You see this in lines 19, 20, and 25. By including the `using` keyword, you are not precluded from continuing to use fully qualified names, as lines 21, 23, and 26 show. There is, however, no need to fully qualify names, because the namespace was included.

This program uses a second routine from the `Console` class called `ReadLine`. As you can see by running this program, the `ReadLine` routine reads what is entered by users up to the time they press Enter. This routine returns what the user enters. In this case, the text entered by the user is assigned with the assignment operator to one of the data members in the name class.

Nested Namespaces

Multiple namespaces can be stored together, and also are stored in a namespace. If a namespace contains other namespaces, you can add them to the qualified name, or you can include the sub-namespace qualified in a `using` statement. For example, the `System` namespace contains several other namespaces, including ones called `Drawing`, `Data`, and `Windows.Forms`. When using classes from these namespaces, you can either qualify these names or you can include them with `using` statements. To include a `using` statement for the `Data` namespace within the `System` namespace, you enter the following:

```
using System.Data;
```

Summary

Today's and tomorrow's lessons are among two of the most important lessons in this book. Classes are the heart of object-oriented programming languages and therefore are the heart and key to C#. In today's lesson, you revisited the concepts of encapsulation, polymorphism, inheritance, and reuse. You then learned how to define the basic structure of a class and how to create data members within your class. You learned one of the first ways to encapsulate your program when you learned how to create properties using the `set` and `get` accessors. The last part of today's lesson introduced you to namespaces and the `using` statement.

Q&A

Q Would you ever use a class with just data members?

A Generally you would not use a class with just data members. The value of a class and of object-oriented programming is the ability to encapsulate both functionality and data into a single package. You learned about only data today. In tomorrow's lesson, you learn how to add the functionality.

Q Should all data members always be declared public so people can get to them?

A Absolutely not! Although many of the data members were declared as public in today's lesson, there are times when you don't want people to get to your data for a number of reasons. One reason is to allow the ability to change the way the data is stored.

Q It was mentioned that there are a bunch of existing classes. How can I find out about these?

A Microsoft provided a bunch of classes called the .NET base classes. Microsoft also has provided documentation on what each of these classes can do. The classes are organized by namespace. At the time this book was written, the only way to get any information on them was through the use of online help. Microsoft included a complete references section for the base classes. You will learn more about the base classes on Day 19 of this book.

Workshop

The Workshop provides quiz questions to help you solidify your understanding of the material covered and exercises to provide you with experience in using what you've learned. Try to understand the quiz and exercise answers before continuing to the next day's lesson. Answers are provided in Appendix A, "Answers."

Quiz

1. What are the four characteristics of an object-oriented program?
2. What two key things can be stored in a class?
3. What is the difference between a data member declared as public and one that hasn't been declared as public?
4. What does adding the keyword `static` do to a data member?
5. What is the name of the application class in Listing 6.2?
6. What commands are used to implement properties?

7. When is `value` used?
8. Is `Console` a class, a data member, a namespace, a routine, or a type?
9. Is `System` a class, a data member, a namespace, a routine, or a type?
10. What keyword is used to include a namespace in a listing?

Exercises

1. Create a class to hold the center of a circle and its radius.
2. Add properties to the `Circle` class created in exercise 1.
3. Create a class that stores an integer called `MyNumber`. Create properties for this number. When the number is stored, multiply it by 100. Whenever it is retrieved, divide it by 100.
4. **BUG BUSTER:** The following program has a problem. Enter it in your editor and compile it. Which lines generate error messages?

```
 1:// A bug buster program
 2:// Is something wrong? Or not?
 3://----------------------------------------
 4:  using System;
 5:  using System.Console;
 6:
 7:  class name
 8:  {
 9:      public string first;
10:  }
11:
12:  class NameApp
13:  {
14:     public static void Main()
15:     {
16:        // Create a name object
17:        name you = new name();
18:
19:        Write("Enter your first name and press enter: ");
20:        you.first = ReadLine();
21:        Write("\nHello {0}!", you.first);
22:     }
23:  }
```

5. Write a class called `die` that will hold the number of sides of a die, `sides`, and the current value of a roll, `value`.
6. Use the class in exercise 5 in a program that declares two dice objects. Set values into the side data members. Set random values into the stored roll values. Note, see Listing 5.3 for help with this program.