HOUR 3

Understanding Objects and Collections

What You'll Learn in This Hour:

- Understanding objects
- Getting and setting properties
- Triggering methods
- Understanding method dynamism
- Writing object-based code
- Understanding collections
- Using the Object Browser

In Hour 1, "Jumping In with Both Feet: A Visual C# 2005 Programming Tour," you were introduced to programming in Visual C# by building a Picture Viewer project. You then spent Hour 2, "Navigating Visual C# 2005," digging into the integrated development environment (IDE) and learning skills critical to your success with Visual C#. In this hour, you're going to start learning about an important programming concept, namely *objects*.

The term *object* as it relates to programming might have been new to you prior to this book. The more you work with Visual C#, the more you'll hear about objects. Visual C# 2005 is a true object-oriented language. This hour isn't going to discuss object-oriented programming in any detail—object-oriented programming is a complex subject and well beyond the scope of this book. Instead, you'll learn about objects in a more general sense.

Everything you use in Visual C# is an object, so understanding this material is critical to your success with Visual C#. For example, forms are objects, as are the controls you place on a form; pretty much every element of a Visual C# project is an object and belongs to a collection of objects. All objects have attributes (called *properties*), most have methods, and many have events. Whether creating simple applications or building large-scale enterprise

solutions, you must understand what an object is and how it works to be successful. In this hour, you'll learn what makes an object an object, and you'll also learn about collections.

By the Way

If you've listened to the programming press at all, you've probably heard the term object oriented, and perhaps words such as polymorphism, encapsulation, and inheritance. In truth, these object-oriented features of Visual C# are exciting, but they're far beyond Hour 3 (or Hour 24, for that matter). You'll learn a little about object-oriented programming in this book, but if you're really interested in taking your programming skills to the next level, you should buy a book dedicated to the subject after you've completed this book.

Understanding Objects

Object-oriented programming has been a technical buzzword for quite some time. Almost everywhere you look—the Web, publications, books—you read about objects. What exactly is an object? Strictly speaking, an *object* is a programming structure that encapsulates data and functionality as a single unit and for which the only public access is through the programming structure's interfaces (properties, methods, and events). In reality, the answer to this question can be somewhat ambiguous because there are so many types of objects—and the number grows almost daily. All objects share specific characteristics, however, such as properties and methods.

The most commonly used objects in Visual C# are the form object and the control object. Earlier hours introduced you to working with forms and controls and even showed you how to set form and control properties. In your Picture Viewer project from Hour 1, for example, you added a picture box and two buttons to a form. Both the PictureBox and the Button controls are *control objects*, but each is a specific type of control object. Another, less-technical example uses pets. Dogs and cats are definitely different entities (objects), but they both fit into the category of Pet objects. Similarly, text boxes and buttons are each a unique type of object, but they're both considered a control object. This small distinction is important.

Understanding Properties

All objects have attributes used to specify and return the state of the object. These attributes are properties, and you've already used some of them in previous hours using the Properties window. Indeed, every object exposes a specific set of properties, but not every object exposes the same set of properties. To illustrate this point, I'll continue with the hypothetical Pet object. Suppose that you have an object, and the

object is a dog. This Dog object has certain properties common to all dogs. These properties include attributes such as the dog's name, the color of its hair, and even the number of legs it has. All dogs have these same properties; however, different dogs have different values for these properties. Figure 3.1 illustrates such a Dog object and its properties.

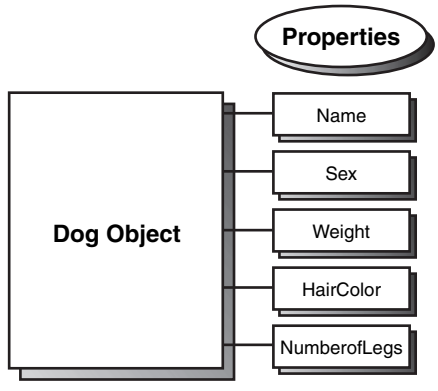

FIGURE 3.1
Properties are the attributes that describe an object.

Getting and Setting Properties

You've already seen how to read and change properties using the Properties window. The Properties window is available only at design time, however, and is used only for manipulating the properties of forms and controls. Most getting and changing of properties you'll perform will be done with Visual C# code, not by using the Properties window. When referencing properties in code, you specify the name of the object first, followed by a period (.), and then the property name as in the following syntax:

```
{ObjectName}.{Property}
```

If you had a Dog object named `Bruno`, for example, you would reference Bruno's hair color this way:

```
Bruno.HairColor
```

This line of code would return whatever value was contained in the `HairColor` property of the Dog object `Bruno`. To set a property to some value, you use an equal sign (=). To change the Dog object Bruno's `Weight` property, for example, you'd use a line of code such as the following:

```
Bruno.Weight = 90;
```

The following line of code places the value of the `Weight` property of the Dog object called `Bruno` into a temporary variable. This statement retrieves the value of the

Weight property because the `Weight` property is referenced on the right side of the equal sign.

```
fltWeight = Bruno.Weight;
```

Variables are discussed in detail in Hour 11, "Using Constants, Data Types, Variables, and Arrays." For now, think of a variable as a storage location. When the processor executes this statement, it retrieves the value in the Weight property of the Dog object Bruno and places it in the variable (storage location) titled `fltWeightVariable`. Assuming that Bruno's Weight is 90, as set in the previous example, the computer would process the code statement like this:

```
fltWeight = 90;
```

Just as in real life, some properties can be read but not changed. Suppose that you have a Sex property to designate the gender of a Dog object. It's impossible for you to change a dog from a male to a female or vice versa (at least I think it is). Because the Sex property can be retrieved but not changed, it's known as a *read-only* property. You'll often encounter properties that can be set in Design view but become read-only when the program is running.

One example of a read-only property is the Height property of the combo box control. Although you can view the value of the Height property in the Properties window, you can't change the value—no matter how hard you try. If you attempt to change the Height property using Visual C# code, Visual C# simply changes the value back to the default—eerie gremlins.

By the Way

The best way to determine which properties of an object are read-only is to consult the online help for the object in question.

Working with an Object and Its Properties

Now that you know what properties are and how they can be viewed and changed, you're going to experiment with properties by modifying the Picture Viewer project you built in Hour 1. Recall from Hour 1 how you learned to set the Height and Width properties of a form using the Properties window. Here, you're going to change the same properties using Visual C# code.

You're going to add two buttons to your Picture viewer. One button will enlarge the form when clicked, whereas the other will shrink the form. This is a simple example, but it illustrates well how to change object properties in Visual C# code.

Start by opening your Picture Viewer project from Hour 1 (you can open the project or the solution file). If you download the code samples from my site, I provide a Picture Viewer project for you to start with.

When the project first runs, the form has the Height and Width you specified in the Properties window. You're going to add buttons to the form that a user can click to enlarge or shrink the form at runtime by following these steps:

1. Double-click frmViewer.cs in the Solutions Explorer window to display the form designer.
2. Add a new button to the form by double-clicking the Button tool in the toolbox. Set the new button's properties as follows:

Property	Set To
`Name`	**btnEnlarge**
`Location`	**342, 261**
`Size`	**21, 23**
`Text`	^ (Note, this is Shift+6)

3. Now for the Shrink button. Again, double-click the Button tool in the toolbox to create a new button on the form. Set this new button's properties as follows:

Property	Set To
`Name`	**btnShrink**
`Location`	**365,261**
`Size`	**21, 23**
`Text`	**v**

Your form should now look like the one in shown in Figure 3.2.

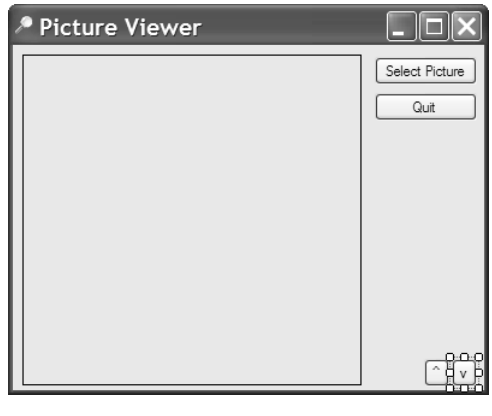

FIGURE 3.2
Each button is an object, as is the form the buttons sit on.

To complete the project, you need to add the small amount of Visual C# code necessary to modify the form's Height and Width properties when the user clicks a button.

4. Access the code for the Enlarge button now by double-clicking the button with the caption ^. Type the following statement exactly as you see it here. Do not press the Enter key or add a space after you've entered this text.

```
this.Width
```

When you type the period, or *dot*, as it's called, a small drop-down list like the one shown in Figure 3.3 appears. Visual C# is smart enough to realize that `this` represents the current form (more on this in a moment), and to aid you in writing code for the object, it gives you a drop-down list containing all the properties and methods of the form. This feature is called *IntelliSense*. When an IntelliSense drop-down box appears, you can use the up and down arrow keys to navigate the list and press Tab to select the highlighted list item. This prevents you from misspelling a member name thereby reducing compile errors. Because Visual C# is fully object-oriented, you'll come to rely on IntelliSense drop-down lists in a big way; I think I'd rather dig ditches than program without them.

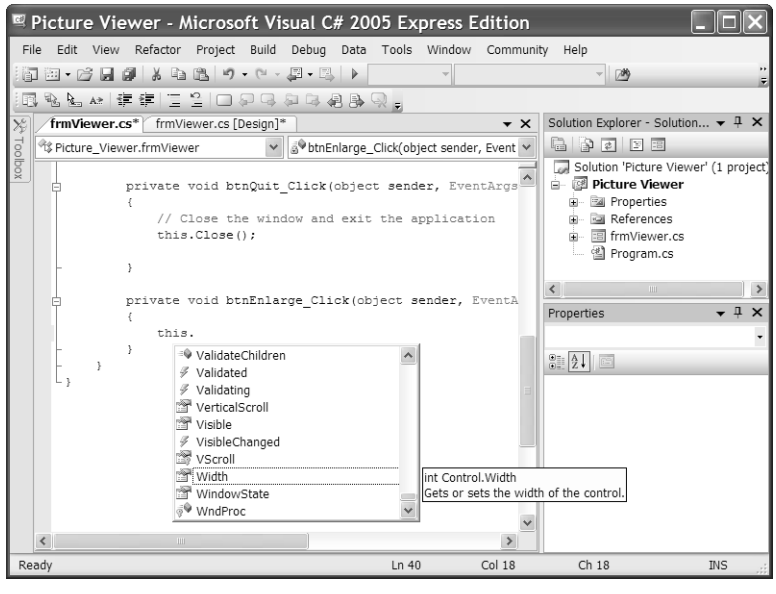

FIGURE 3.3 IntelliSense drop-down lists (also called *auto-completion drop-down lists*) make coding dramatically easier.

5. Use the Backspace key to completely erase the code you just entered and enter the following code in its place (press Enter at the end of each line):

```
this.Width = this.Width + 20;
this.Height = this.Height + 20;
```

By the Way

Notice that all Visual C# statements end with a semicolon. This semicolon is required, and it lets the Visual C# compiler know that it has reached the end of a statement.

Remember from before that the word `this` refers to the object to which the code belongs (in this case, the form). `this` is a *reserved* word; it's a word that you can't use to name objects or variables because Visual C# has a specific meaning for it. When writing code within a form module, as you're doing here, always use the reserved word `this` rather than the name of the form. `this` is much shorter than using the full name of the current form, and it makes the code more portable (you can copy and paste the code into another form module and not have to change the form name to make the code work). Also, should you change the name of the form at any time in the future, you won't have to change references to the old name.

The code you've entered does nothing more than set the Width and Height properties of the form to whatever the current value of the Width and Height properties happens to be, plus 20 pixels.

6. Redisplay the form designer by selecting the tab named frmViewer.cs [Design] at the top of the designer window. Then double-click the button with the caption v to access its Click event and add the following code:

```
this.Width = this.Width - 20;
this.Height = this.Height - 20;
```

This code is similar to the code in the `btnEnlarge_Click` event, except that it reduces the Width and Height properties of the form by 20 pixels. Your screen should now look like Figure 3.4.

Did you Know?

As you create projects, it's a good idea to save frequently. Save your project now by clicking the Save All button on the toolbar.

Once again, display the form designer by clicking the tab frmViewer.cs [Design]. Your Properties Example project is now ready to be run! Press F5 to put the project in Run mode. Before continuing, click the Select Picture button and choose a picture from your hard drive.

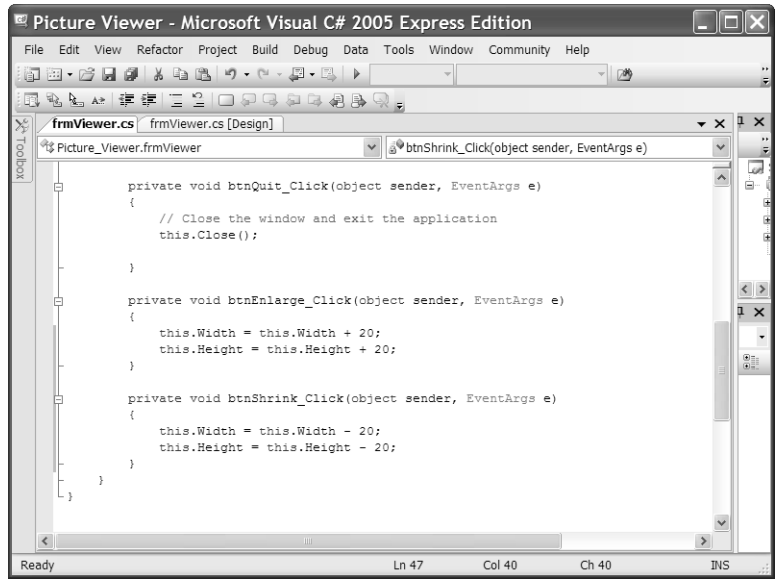

FIGURE 3.4
The code you've entered should look exactly like this.

Next, click the ^ button a few times and notice how the form gets bigger. The form will grow bigger (see Figure 3.5).

FIGURE 3.5
What you see is what you get—the form you created should look just as you designed it.

Next, click the v button to make the form smaller. When you've clicked enough to satisfy your curiosity (or until you get bored), end the running program and return to Design mode by clicking the Stop Debugging button on the toolbar.

Did you notice how the buttons and the image on the form didn't resize as the form's size was changed? In Hour 6, "Building Forms—Advanced Techniques," you'll learn how to make your forms resize their contents.

Understanding Methods

In addition to properties, most objects have *methods*. Methods are actions the object can perform, in contrast to attributes that describe the object. To understand this distinction, think about the Pet object example. A Dog object has a certain set of actions that it can perform. These actions, called methods in Visual C#, include barking, tail wagging, and chewing carpet (don't ask). Figure 3.6 illustrates the Dog object and its methods.

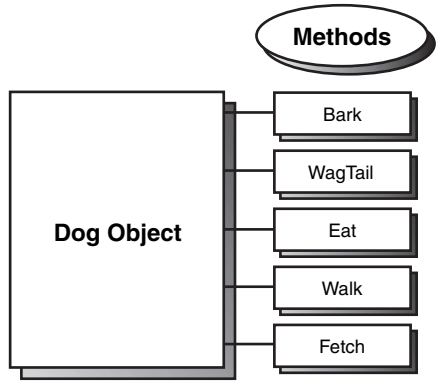

FIGURE 3.6
Invoking a method causes the object to perform an action.

Triggering Methods

Think of methods as functions—which is exactly what they are. When you invoke a method, code is executed. You can pass data to a method, and methods can return values. However, a method is neither required to accept *parameters* (data passed to it by the calling code) nor to return a value; many methods simply perform an action in code. Invoking (triggering) a method is similar to referencing the value of a property: You first reference the object's name, and then a dot, and then the method name as shown next:

```
{ObjectName}.{Method}
```

For example, to make the hypothetical Dog object `Bruno` bark using Visual C# code, you would use this line of code:

```
Bruno.Bark();
```

By the Way

Methods calls in Visual C# must always have parentheses. Sometimes they'll be empty, but at other times they'll contain data to pass to the method.

By the Way

Methods are generally used to perform an action using an object, such as saving or deleting a record in a database. Properties, on the other hand, are used to get and set attributes of the object. One way to tell in code whether a statement is a property reference or method call is that a method call will have a set of parentheses after it, as in `frmAlbum.ShowDialog();`.

Invoking methods is simple; the real skill lies in knowing what methods an object supports and when to use a particular method.

Understanding Method Dynamism

Properties and methods go hand in hand, and at times a particular method might become unavailable because of one or more property values. For example, if you were to set the NumberofLegs property on the Dog object Bruno equal to zero, the `Walk()` and `Fetch()` methods would obviously be inapplicable. If you were to set the `NumberofLegs` property back to four, you could then trigger the `Walk()` or `Fetch()` method again.

Building a Simple Object Example Project

The only way to really grasp what objects are and how they work is to use them.

Every project you've built so far uses objects, but you're now going to create a sample project that specifically illustrates using objects. If you're new to programming with objects, you'll probably find this a bit confusing. However, I'll walk you through step-by-step, explaining each section in detail.

You're going to modify your Picture Viewer project to include a button that, when clicked, draws a colored border around the picture.

By the Way

In Hour 18, "Working with Graphics," you'll learn all about the drawing functionality within Visual C#.

Creating the Interface for the Drawing Project

Continuing on with the Picture Viewer project you've been using in this chapter, add a new button to the form and set its properties as shown in the following table:

Property	Value
Name	**btnDrawBorder**
Location	**301, 69**
Size	**85, 23**
Text	**Draw Border**

Writing the Object-Based Code

You're now going to add code to the Click event of the button. I'm going to explain each statement, and at the end of the steps, I'll show the complete code listing.

1. Double-click the Draw Border button to access its Click event.
2. Enter the first line of code as follows (remember to press Enter at the end of each statement):

```
Graphics objGraphics = null;
```

Here you've just created a variable that will hold an instance of an object. Objects don't materialize out of thin air; they have to be created. When a form is loaded into memory, it loads all its controls (that is, creates the control objects), but not all objects are created automatically like this. The process of creating an instance of an object is called *instantiation*. When you load a form, you instantiate the form object, which in turn instantiates its control objects. You could load a second instance of the form, which in turn would instantiate a new instance of the form and new instances of all controls. You would then have two forms in memory, and two of each used control.

To instantiate an object in code, you create a variable that holds a reference to an instantiated object. You then manipulate the variable as an object. The variable declaration statement you wrote in step 2 creates a new variable called `objGraphics`, which holds a reference to an object of type Graphics (the type comes first, then the variable name). You learn more about variables in Hour 18.

Next, enter the second line of code exactly as shown here:

```
objGraphics = this.CreateGraphics();
```

`CreateGraphics()` is a method of the form (remember, the keyword `this` is shorthand for referencing the current form). Under the hood, the `CreateGraphics()` method is pretty complicated, and I discuss it in detail in Hour 18. For now, understand that the method `CreateGraphics()` instantiates a new object that represents the client area of the current form. The client area is the gray area within the borders and title bar of a form. Anything drawn onto the `objGraphics()` object will appear on the form. What you've done is set the variable `objGraphics()` to point to an object that was returned by the `CreateGraphics()` method. Notice how values returned by a property or method don't have to be traditional values such as numbers or text; they could also be objects.

Enter the third line of code as shown next:

```
objGraphics.Clear(SystemColors.Control);
```

This statement clears the background of the form using whatever color the user has selected as the Windows Control color, which Windows uses to paint forms.

How does this happen? After declaring the `objGraphics` object, you used the `CreateGraphics()` method of the form to instantiate a new graphics object in the variable `objGraphics()`. With the code statement you just entered, you're calling the `Clear()` method of the `objGraphics()` object. The `Clear()` method is a method of all Graphics objects used to clear the graphic surface. The `Clear()` method accepts a single parameter: the color you want used to clear the surface.

The value you're passing to the parameter might seem a bit odd. Remember that "dots" are a way of separating objects from their properties and methods (properties, methods, and events are often called object *members*). Knowing this, you can discern that `SystemColors` is an object because it appears before any of the dots. Object references can and do go pretty deep, and you'll use many dots throughout your code. The key points to remember are

- Text that appears to the left of a dot is always an object (or namespace).
- Text that appears to the right of a dot is a property reference or method call. If the text is followed by a set of parentheses (), it's a method call. If not, it's most likely a property.
- Methods can return objects, just as properties can. The only surefire ways to know whether the text between two dots is a property or method is to look at the icon of the member in the IntelliSense drop-down or to consult the documentation of the object.

The final text in this statement is the word `Control`. Because `Control` isn't followed by a dot, you know that it's not an object; therefore, it must be a property or method. Because you expect this string of object references to return a color value to be used to clear the graphic object, you know that `Control` in this instance must be a property or a method that returns a value (because you need the return value to set the `Clear()` method). A quick check of the documentation would tell you that `Control` is indeed a property, but you don't even need to do that because there are no parenthesis at the end of `Control`, so it *can't* be a method and therefore has to be a property. The value of `Control` always equates to the color designated on the user's computer for the face of forms and buttons. By default, this is a light gray (often fondly referred to as *battleship gray*), but users can change this value on their computers. By using this property to specify a color rather than supplying the actual value for gray, you're assured that no matter the color scheme used on a computer, the code will clear the form to the proper system color. System colors are explained in Hour 18.

Enter the following statement. Note: Press Enter after each line. Remember, Visual C# uses a semicolon to denote the end of a statement, so it will consider all three lines as being one code statement.

```
objGraphics.DrawRectangle(Pens.Blue,
      picShowPicture.Left - 1, picShowPicture.Top - 1,
      picShowPicture.Width + 1, picShowPicture.Height + 1);
```

This statement draws a blue rectangle around the picture on the form. Within this statement is a single method call and five property references. Can you tell what's what? Immediately following `objGraphics` (and a dot) is `DrawRectangle`. Because no equal sign is present, you can deduce that this is a method call. As with the `Clear()` method, the parentheses after `DrawRectangle` are used to enclose values passed to the method.

The `DrawRectangle()` method accepts the following parameters in the order in which they appear here:

- A pen
- X value of the upper-left corner
- Y value of the upper-left corner
- Width of the rectangle
- Height of the rectangle

The `DrawRectangle()` method draws a prefect rectangle using the X, Y, Width, and Height values passed to it. The attributes of the line (color, width, and so on) are

determined by the pen specified in the `Pen` parameter. I'm not going to go into detail on pens here (see Hour 18). Looking at the dots once more, notice that you're passing the `Blue` property of the Pens object. Blue is an object property that returns a predefined `Pen` object that has a width of 1 pixel and the color blue.

For the next two parameters, you're passing property values. Specifically, you're passing the top and left values for the picture, less one. If you passed the exact left and top values, the rectangle would be drawn on the form at exactly the top and left properties of the PictureBox, and you wouldn't see them because controls by default overlap any drawing performed on the form.

The last two property references are for the Height and Width of the PictureBox. Again, we adjust the values by one to ensure that the rectangle is drawn outside the borders of the PictureBox.

Finally, you have to clean up after yourself by entering the following code statement:

```
objGraphics.Dispose();
```

Objects often use other objects and resources. The underlying mechanics of an object can be truly mind boggling and are almost impossible to discuss in an entry-level programming book. The net effect, however, is that you must explicitly destroy most objects when you're finished with them. If you don't destroy an object, it might persist in memory, and it might hold references to other objects or resources that exist in memory. This means that you can create a *memory leak* within your application that slowly (or rather quickly) munches system memory and resources. This is one of the cardinal no-no's of Windows programming, yet the nature of using resources and the fact you're responsible for telling your objects to clean up after themselves makes this easy to do. If your application causes memory leaks, your users won't call for a plumber, but they might reach for a monkey wrench….

Objects that must explicitly be told to clean up after themselves usually provide a `Dispose()` method. When you're finished with such an object, call `Dispose()` on the object to make sure that it frees any resources it might be holding.

For your convenience, here are all the lines of code:

```
Graphics objGraphics = null;
objGraphics = this.CreateGraphics();
objGraphics.Clear(SystemColors.Control);
objGraphics.DrawRectangle(Pens.Blue,
      picShowPicture.Left - 1, picShowPicture.Top - 1,
      picShowPicture.Width + 1, picShowPicture.Height + 1);
objGraphics.Dispose();
```

Click Save All on the toolbar to save your work before continuing.

Testing Your Object Example Project

Now the easy part: Run the project by pressing F5 or by clicking the Start button on the toolbar. Your form looks pretty much like it did at design time. Clicking the button causes a blue rectangle to be drawn around the PictureBox (see Figure 3.7).

By the Way

If you receive any errors when you attempt to run the project, go back and make sure that the code you entered exactly matches the code I've provided.

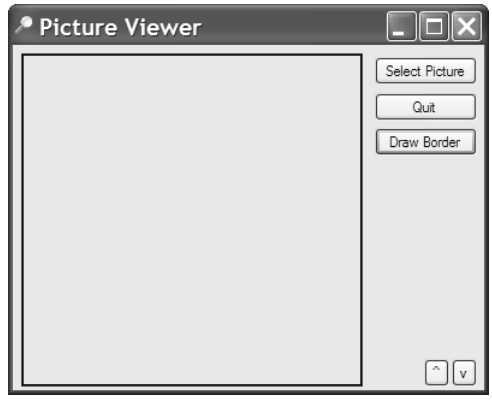

FIGURE 3.7
Simple lines and complex drawings are accomplished using objects.

By the Way

If you use Alt+Tab to switch to another application after drawing the rectangle, the rectangle will be gone when you come back to your form. In fact, this will occur anytime you overlay the graphics with another form. In Hour 18, you'll learn why this is so and how to work around this behavior.

Stop the project now by clicking Stop Debugging on the Visual C# toolbar. What I hope you've gained from building this example is not necessarily that you can now draw a rectangle (which is cool), but rather an understanding of how objects are used in programming. As with learning almost anything, repetition aids in understanding. That said, you'll be working with objects *a lot* throughout this book.

Understanding Collections

A *collection* is just what its name implies: a collection of objects. Collections make it easy to work with large numbers of similar objects by enabling you to create code that performs iterative processing on items within the collection. *Iterative processing* is an operation that uses a loop to perform actions on multiple objects, rather than writing the operative code for each object. In addition to containing an indexed set

of objects, collections also have properties and might have methods. Figure 3.8 illustrates the structure of a collection.

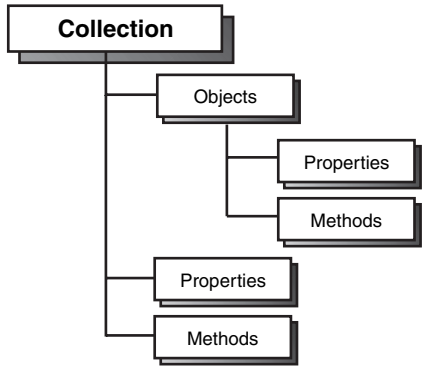

FIGURE 3.8
Collections contain sets of like objects, and they have their own properties and methods.

Continuing with the Dog/Pet object metaphor, think about what an Animals collection might look like. The Animals collection might contain one or more Pet objects, or it might be empty (contain no objects). All collections have a `Count` property that returns the total count of objects contained within the collection. Collections might also have methods, such as a `Delete()` method used to remove objects from the collection and an `Add()` method used to add a new object to the collection.

To better understand collections, you're going to create a small Visual C# project that cycles through the Controls collection of a form and tells you the value of the `Name` property of every control on the form. To create your sample project, follow these steps:

1. Start Visual C# now (if it's not already loaded) and create a new Windows Application project titled **Collections Example**.
2. Rename the form from Form1.vs to **frmCollectionsExample.cs** using the Solution Explorer. If prompted to update all code references to use the new name, select Yes. Next, set the form's Text property to **Collections Example** (you will need to click the form once to display its properties).
3. Add a new button to the form by double-clicking the Button tool in the toolbox. Set the button's properties as follows:

Property	Value
`Name`	**btnShowNames**
`Location`	**88,112**
`Size`	**120,23**
`Text`	**Show Control Names**

4. Next, add some text box and button controls to the form. As you add the controls to the form, be sure to give each control a unique name. Feel free to use any name you want, but you can't use spaces in a control name. You might want to drag the controls to different locations on the form so that they don't overlap.

5. When you're finished adding controls to your form, double-click the Show Control Names button to add code to its Click event. Enter the following code:

```
for (int intIndex = 0; intIndex < this.Controls.Count; intIndex++)
   {
      MessageBox.Show("Control #" + intIndex.ToString() +
            " has the name " + this.Controls[intIndex].Name);
   }
```

By the Way

Every form has a Controls collection, which might not contain any controls. Even if no controls are on the form, the form still has a Controls collection.

The first statement (the one that begins with `for`) accomplishes a few tasks. First, it initializes the variable intIndex to 0, and then tests the variable. It also starts a loop executing the statement block (loops are discussed in Hour 14, "Looping for Efficiency"), incrementing intIndex by one until intIndex equals the number of controls on the form, less one. The reason that intIndex must always be less than the Count property is that when referencing items in a collection, the first item is always item zero—collections are zero-based. Thus, the first item is in location zero, the second item is in location one, and so forth. If you tried to reference an item of a collection in the location of the value of the Count property, an error would occur because you would be referencing an index that is one higher than the actual locations within the collection.

The MessageBox.Show() method (discussed in detail in Hour 17, "Interacting with Users") is a class available in the .NET Framework that is used to display a simple dialog box with text. The text that you are providing, which the `MessageBox.Show()` method will display, is a concatenation of multiple strings of text. (*Concatenation* is the process of adding strings together; it is discussed in Hour 12, "Performing Arithmetic, String Manipulation, and Date/Time Adjustments.")

Run the project by pressing F5 or by clicking Start on the toolbar. Ignore the additional controls that you placed on the form and click the Show Control Names button. Your program then displays a message box similar to the one shown in Figure 3.9 for each control on your form (because of the loop). When the program is finished displaying the names of the controls, choose Stop Debugging from the Debug menu to stop the program and then save the project.

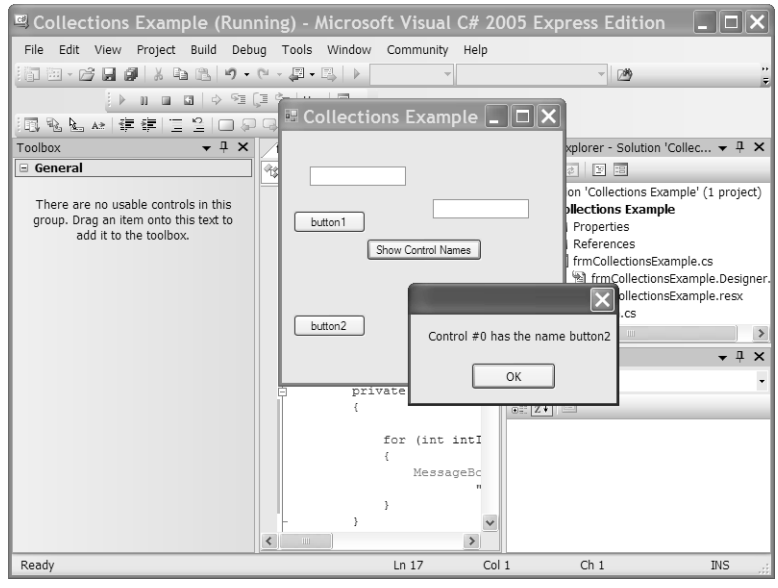

FIGURE 3.9
The Controls collection enables you to get to each and every control on a form.

Because everything in Visual C# 2005 is an object, you can expect to use numerous collections as you create your programs. Collections are powerful, and the quicker you become comfortable using them, the more productive you'll be.

Using the Object Browser

Visual C# 2005 includes a useful tool that enables you to easily view members (properties, methods, and events) of all the objects in a project: the Object Browser (see Figure 3.10). This is useful when dealing with objects that aren't well documented because it enables you to see all the members an object supports. To view the Object Browser, choose View, Other Windows, Object Browser from the menu.

The Browse drop-down list in the upper-left corner of the Object Browser is used to determine the *browsing scope*. You can choose My Solution to view only the objects referenced in the active solution, or you can choose All Components to view all possible objects. You can customize the object set by clicking the drop-down arrow next to the Object Browser Settings button to the far right of the Browse drop-down list. I don't recommend changing the custom object setting until you have some experience using Visual C# objects as well as experience using the Object Browser.

The top-level nodes (each item in the tree is referred to as a node) in the Objects tree are libraries. *Libraries* are usually DLL or EXE files on your computer that contain one or more objects. To view the objects within a library, simply expand the library

node. As you select objects within a library, the list to the right of the Objects tree shows information regarding the members of the selected object (refer to Figure 3.10). For even more detailed information, click a member in the list on the right, and the Object Browser shows information about the member in the area below the two lists.

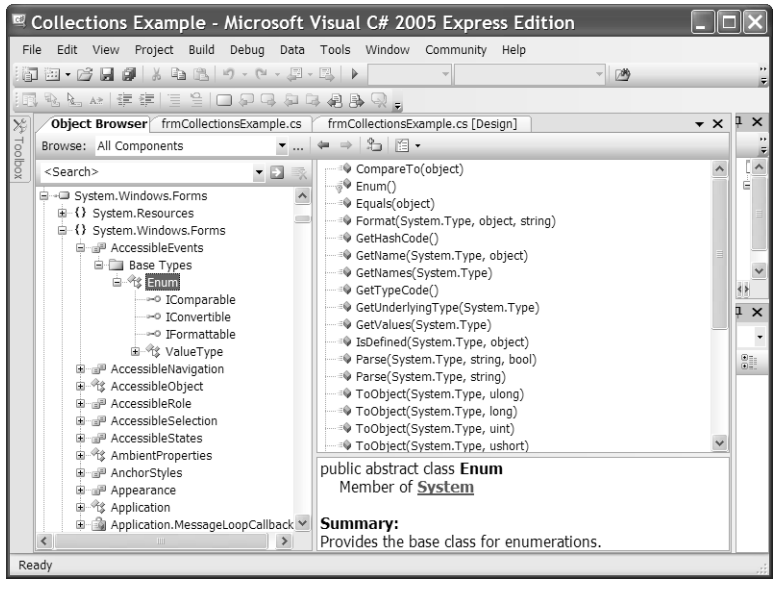

FIGURE 3.10 The Object Browser enables you to view all properties and methods of an object.

Summary

In this hour, you learned a lot about objects. You learned how objects have properties, which are attributes that describe the object. Some properties can be set at design time using the Properties window, and most can also be set at runtime in Visual C# code. You learned that referencing a property on the left side of the equal sign has the effect of changing a property, whereas referencing a property on the right side of the equal sign retrieves a property's value.

In addition to properties, you learned that objects have executable functions, called *methods*. Like properties, methods are referenced by using a dot at the end of an object reference. An object might contain many methods and properties, and some properties can even be objects themselves. You learned how to "follow the dots" to interpret a lengthy object reference.

Objects are often used as a group, called a *collection*. You learned that a collection often contains properties and methods, and that collections let you easily iterate

through a set of like objects. Finally, you learned that the Object Browser can be used to explore all the members of an object in a project.

The knowledge you've gained in this hour is fundamental to understanding programming with Visual C# because objects and collections are the basis on which applications are built. After you have a strong grasp of objects and collections—and you will have by the time you've completed all the hours in this book—you'll be well on your way to fully understanding the complexities of creating robust applications using Visual C# 2005.

Q&A

Q. ***Is there an easy way to get help about an object's member?***

A. Absolutely. Visual C#'s context-sensitive Help extends to code as well as to visual objects. To get help on a member, write a code statement that includes the member (it doesn't have to be a complete statement), position the cursor within the member text, and press F1. For instance, to get help on the `int` data type, you could type **`int`**, position the cursor within the word `int`, and press F1.

Q. ***Are there any other types of object members besides properties and methods?***

A. Yes. An event is actually a member of an object, although it's not always thought of that way. Although not all objects support events, most objects do support properties and methods.

Workshop

The Workshop is designed to help you anticipate possible questions, review what you've learned, and get you thinking about how to put your knowledge into practice.

Quiz

1. True or False: Visual C# .NET is a true object-oriented language.
2. An attribute that defines the state of an object is called a _____.
3. To change the value of a property, the property must be referenced on which side of an equal sign?

4. What is the term for when a new object is created from a template?
5. An external function of an object (one that is available to code using an object) is called a _____.
6. True or False: A property of an object can be another object.
7. A group of like objects is called a _____.
8. What tool is used to explore the members of an object?

Answers

1. True
2. Property
3. The left side
4. Instantiation
5. Method
6. True
7. Collection
8. The Object Browser

Exercises

1. Create a new project and add two text boxes and a button to the form. Write code that, when a button is clicked, places the text in the first text box into the second text box. Hint: Use the Text property of the text box controls.
2. Modify the collections example in this hour to print the height of all controls, rather than the name.