James Foxall

Sams Teach Yourself

Visual Basic® 2012

in 24 Hours

SAMS

FREE SAMPLE CHAPTER

SHARE WITH OTHERS

James Foxall

Sams **Teach Yourself**

Visual Basic 2012

in 24 Hours

SAMS 800 East 96th Street, Indianapolis, Indiana, 46240 USA

Sams Teach Yourself Visual Basic 2012 in 24 Hours, Complete Starter Kit

ISBN-13: 978-0-672-33629-4

ISBN-10: 0-672-33629-4

Library of Congress Cataloging-in-Publication Data is on file.

Printed in the United States of America

First Printing October 2012

Trademarks

Warning and Disclaimer

Bulk Sales

Sams Publishing offers excellent discounts on this book when ordered in quantity for bulk purchases or special sales. For more information, please contact

U.S. Corporate and Government Sales
1-800-382-3419
corpsales@pearsontechgroup.com

For sales outside of the U.S., please contact

International Sales
international@pearsoned.com

Editor-in-Chief
Greg Wiegand

Executive Editor
Neil Rowe

Development Editor
Mark Renfrow

Managing Editor
Sandra Schroeder

Project Editor
Mandie Frank

Copy Editor
Margo Catts

Indexer
Cheryl Lenser

Proofreader
Sarah Kearns

Technical Editor
J. Boyd Nolan

Publishing Coordinator
Cindy Teeters

Multimedia Developer
Tim Warner

Designer
Gary Adair

Composition
TnT Design, Inc.

Contents at a Glance

Table of Contents

About the Author

James Foxall is president of Tigerpaw Software, Inc. (www.tigerpawsoftware.com), a Bellevue, Nebraska, Microsoft Certified Partner specializing in software solutions for technology providers. Tigerpaw's award-winning business automation solution is designed to automate contact management, marketing, service and repair, proposal generation, inventory control, and purchasing. At the start of 2012, the current release of Tigerpaw had more than 35,000 licensed users. Foxall's experience in creating certified Office-compatible software has made him an authority on application interface and behavior standards of applications for the Microsoft Windows and Microsoft Office environments.

Foxall has been writing commercial production Visual Basic code for more than 14 years. He's the author of numerous books, including *Practical Standards for Microsoft Visual Basic* and *MCSD in a Nutshell: The Visual Basic Exams*. He also has written articles for *Access-Office-VBA Advisor* and *Visual Basic Programmer's Journal*. Foxall has a bachelor's degree in management of information systems (MIS) and a master's degree in Business Administration (MBA). He is a Microsoft Certified Solution Developer and an international speaker on programming technologies as well as business process improvements. James enjoys spending time with his family, playing guitar, listening to amazing bands, and playing computer games. You can reach him at www.jamesfoxall.com.

Dedication

This book is dedicated to all of my great co-workers at Tigerpaw; thank you for your dedication and your passion!

Acknowledgments

I would like to thank all the great people at Sams for their input and hard work; this book would not be possible without them!

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

You can email or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book, and that due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or email address. I will carefully review your comments and share them with the author and editors who worked on the book.

Email: consumer@samspublishing.com

Mail: Neil Rowe
Executive Editor
Sams Publishing
800 East 96th Street
Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at www.samspublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

Introduction

Visual Basic 2012 is Microsoft's latest incarnation of the enormously popular Visual Basic language, and it's fundamentally different from the versions that came before it. Visual Basic is more powerful and more capable than ever before, and its features and functionality are on par with "higher-level" languages such as C++. One consequence of this newfound power is added complexity. Gone are the days when you could sit down with Visual Basic and the online Help and teach yourself what you needed to know to create a functional program.

Audience and Organization

This book is targeted toward those who have little or no programming experience or who might be picking up Visual Basic as a second language. The book has been structured and written with a purpose: to get you productive as quickly as possible. I've used my experiences in writing large commercial applications with Visual Basic and teaching Visual Basic to create a book that I hope cuts through the fluff and teaches you what you need to know. All too often, authors fall into the trap of focusing on the technology rather than on the practical application of the technology. I've worked hard to keep this book focused on teaching you practical skills that you can apply immediately to a development project. Feel free to post your suggestions or success stories at www.jamesfoxall.com/forums.

This book is divided into five parts, each of which focuses on a different aspect of developing applications with Visual Basic. These parts generally follow the flow of tasks you'll perform as you begin creating your own programs with Visual Basic. I recommend that you read them in the order in which they appear:

- Part I, "The Visual Basic 2012 Environment," teaches you about the Visual Basic environment, including how to navigate and access Visual Basic's numerous tools. In addition, you'll learn about some key development concepts such as objects, collections, and events.
- Part II, "Building a User Interface," shows you how to build attractive and functional user interfaces. In this part, you'll learn about forms and controls—the user interface elements such as text boxes and list boxes.
- Part III, "Making Things Happen—Programming," teaches you the nuts and bolts of Visual Basic 2012 programming—and there's a lot to learn. You'll

discover how to create modules and procedures, as well as how to store data, perform loops, and make decisions in code. After you've learned the core programming skills, you'll move into object-oriented programming and debugging applications.

- Part IV, "Working with Data," introduces you to working with graphics, text files, and programming databases and shows you how to automate external applications such as Word and Excel. In addition, this part teaches you how to manipulate a user's file system and the Windows Registry.
- Part V, "Deploying Solutions and Beyond," shows you how to distribute an application that you've created to an end user's computer. In Hour 24, "The 10,000-Foot View," you'll learn about Microsoft's .NET initiative from a higher, less-technical level.

Many readers of previous editions have taken the time to give me input on how to make this book better. Overwhelmingly, I was asked to have examples that build on the examples in the previous chapters. In this book, I have done that as much as possible. Instead of learning concepts in isolated bits, you'll be building a feature-rich Picture Viewer program throughout the course of this book. You'll begin by building the basic application. As you progress through the chapters, you'll add menus and toolbars to the program, build an Options dialog box, modify the program to use the Windows Registry and a text file, and even build a setup program to distribute the application to other users. I hope you find this approach beneficial in that it enables you to learn the material in the context of building a real program.

Conventions Used in This Book

This book uses several design elements and conventions to help you prioritize and reference the information it contains:

By the Way

By the Way boxes provide useful sidebar information that you can read immediately or circle back to without losing the flow of the topic at hand.

Did you Know?

Did You Know? boxes highlight information that can make your Visual Basic programming more effective.

Watch Out!

Watch Out! boxes focus your attention on problems or side effects that can occur in specific situations.

New terms appear in an *italic* typeface for emphasis.

In addition, this book uses various typefaces to help you distinguish code from regular English. Code is presented in a `monospace` font. Placeholders—words or characters that represent the real words or characters you would type in code—appear in *`italic monospace`*. When you are asked to type or enter text, that text appears in **bold**.

Menu options are separated by a comma. For example, when you should open the File menu and choose the New Project menu option, the text says "Select File, New Project."

Some code statements presented in this book are too long to appear on a single line. In these cases, a line-continuation character (an underscore) is used to indicate that the following line is a continuation of the current statement.

Onward and Upward!

This is an exciting time to be learning how to program. It's my sincerest wish that when you finish this book, you feel capable of using many of Visual Basic's tools to create, debug, and deploy modest Visual Basic programs. Although you won't be an expert, you'll be surprised at how much you've learned. And I hope this book will help you determine your future direction as you proceed down the road to Visual Basic mastery.

I love programming with Visual Basic, and sometimes I find it hard to believe I get paid to do so. I hope you find Visual Basic as enjoyable as I do!

This page intentionally left blank

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 Basic 2012 Programming Tour," you were introduced to programming in Visual Basic by building a Picture Viewer project. You then spent Hour 2, "Navigating Visual Basic 2012," digging into the integrated development environment (IDE) and learning skills critical to your success with Visual Basic. In this hour, you begin learning about an important programming concept: *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 Basic, the more you'll hear about objects. Visual Basic 2012, unlike its early predecessors, is a true object-oriented language. This hour doesn't 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 Basic is an object, so understanding this material is critical to your success with Visual Basic. For example, forms are objects, as are the controls you place on a form. Pretty much every element of a Visual Basic 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. 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 Basic are exciting, but they're far beyond this hour (or the last hour, 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, but as far as Visual Basic programmers are concerned, it became a reality only with Visual Basic .NET. (No previous version of Visual Basic was a true object-oriented language.) 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 Basic 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, a text box and a button is each a unique type of object, but they're both considered control objects. This small distinction is important.

Understanding Properties

All objects have attributes that are 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 in 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.

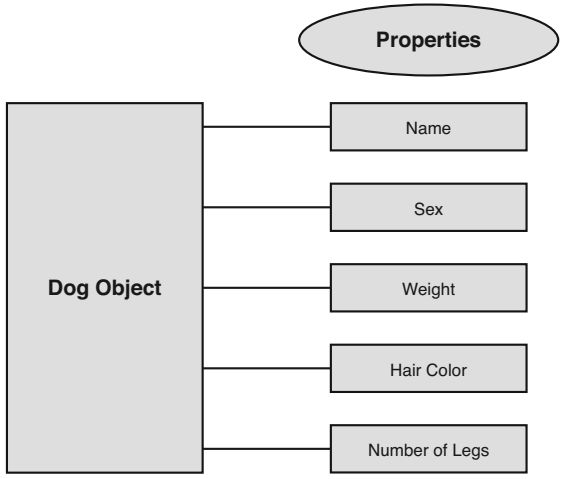

FIGURE 3.1 Properties are the attributes that describe an object.

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 to manipulate the properties of forms and controls. Most getting and setting of properties you'll perform will be done with Visual Basic code, not in the Properties window. When referencing properties in code, you specify the object's name first, followed by a period (.), and then the property name, as in the following syntax:

```
ObjectName.Property
```

If you had a `Button` object named `btnClickMe`, for example, you would reference the button's `Text` property this way:

```
btnClickMe.Text
```

This line of code would return whatever value was contained in the `Text` property of the `Button` object `btnClickMe`. To set a property to some value, you use an equals sign (=). To change the `Button` object's `Left` property, for example, you'd use a line of code such as the following:

```
btnClickMe.Left = 90
```

When you reference a property on the left side of an equals sign, you're setting the value. When you reference a property on the right side of the equals sign, you're getting (reading) the value. In the early days of BASIC, you actually used the word `Let` when setting values. This made the code easier to read for novices, but it was unnecessarily tedious. Nevertheless, using `Let` makes the statement clearer for this example, so I'll show the same code statement as before with the word `Let`:

```
Let btnClickMe.Left = 90
```

It's easier to see here that referencing the property on the left side of the equals sign indicates that you're setting the property to some value. The keyword `Let` is no longer a valid way to make variable assignments. If you enter a code statement that uses `Let`, you won't receive an error, but the code editor (also known as *the gremlins*) automatically removes the word `Let` from the statement for you.

The following line of code places the value of the `Left` property of the `Button` object called `btnClickMe` into a temporary variable. This statement retrieves the value of the `Left` property because the `Left` property is referenced on the right side of the equals sign:

```
intLeftVariable = btnClickMe.Left
```

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 `Left` property of the `Button` object `btnClickMe` and places it in the variable (storage location) titled `intLeftVariable`. Assuming that `btnClickMe`'s `Left` property value is 90, as set in the previous example, the computer would process the code statement like this:

```
intLeftVariable = 90
```

Just as in real life, some properties can be read but not changed. Think back to the hypothetical pet object, and suppose that you have a `Gender` 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 `Gender` 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 Basic code, Visual Basic 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'll 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 in the Properties window. Here, you'll change the same properties, now using Visual Basic code.

You'll add two buttons to your Picture Viewer. One button will enlarge the form when clicked, and the other will shrink the form. This is a simple example, but it illustrates well how to change object properties in Visual Basic code.

Start by opening your Picture Viewer project from Hour 1. If you download the code samples from my site, I provide a Picture Viewer project for you to start with. Double-click `ViewerForm.vb` in the Solution Explorer window to show the form designer.

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

1. 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	Value
Name	btnEnlarge
Location	338,261
Size	21,23
Text	^ (Note: This is Shift+6.)

2. 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	Value
Name	btnShrink
Location	359,261
Size	21,23
Text	v

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

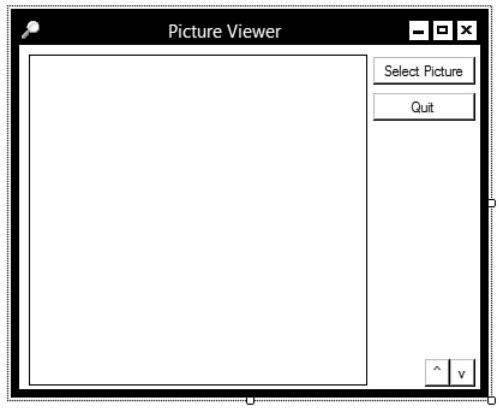

FIGURE 3.2
Each button is an object, as is the form on which the buttons sit.

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

3. Access the code for the Enlarge button by double-clicking the ^ button. 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:

```
Me.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 Basic is smart enough to realize that `Me` represents the current form (more on this in a moment). To help you write 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 Basic 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.

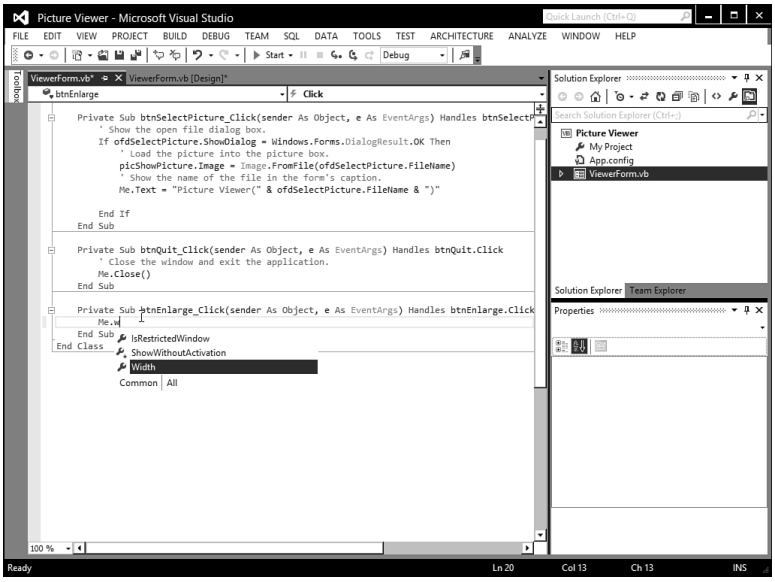

FIGURE 3.3
IntelliSense drop-down lists (also called auto-completion drop-down lists) make coding dramatically easier.

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

```
Me.Width = Me.Width + 20
Me.Height = Me.Height + 20
```

Remember from before that the word `Me` doesn't refer to a person; it refers to the object to which the code belongs (in this case, the form). `Me` is a *reserved* word; it's a word that you can't use to name objects or variables because Visual Basic has a specific meaning for it. When writing code within a form module, as you're doing here, always use the reserved word `Me` rather than the name of the form. `Me` 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 just entered does nothing more than set the form's `Width` and `Height` properties to their current value plus 20 pixels.

5. Redisplay the form designer by selecting the tab named ViewerForm.vb [Design] at the top of the designer window. Then double-click the button with the v to access its `Click` event, and add the following code:

```
Me.Width = Me.Width - 20
Me.Height = Me.Height - 20
```

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

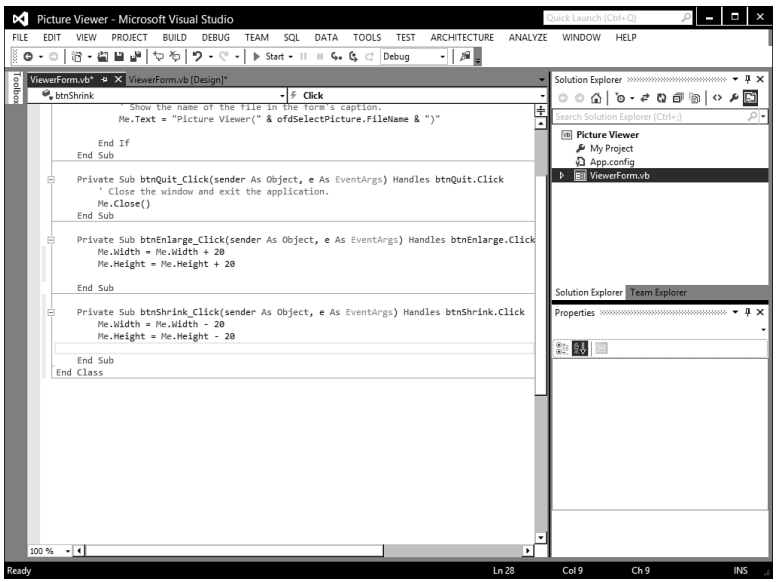

FIGURE 3.4
The code you've entered should look exactly like this.

Did you Know?

As you create projects, it's a good idea to save frequently. When an asterisk appears to the right of a tab's title (as you can see in Figure 3.4), you know that the file edited within that tab has been changed but not saved. Save your project now by clicking the Save All button on the toolbar.

Again, display the form designer by clicking the tab ViewerForm.vb [Design]. Your Properties Example project is now ready to be run! Follow these steps to test your changes:

1. Press F5 to put the project in Run mode.
2. Click the Select Picture button, and choose a picture from your hard drive.
3. Click the ^ button a few times and notice how the form gets bigger (see Figure 3.5).

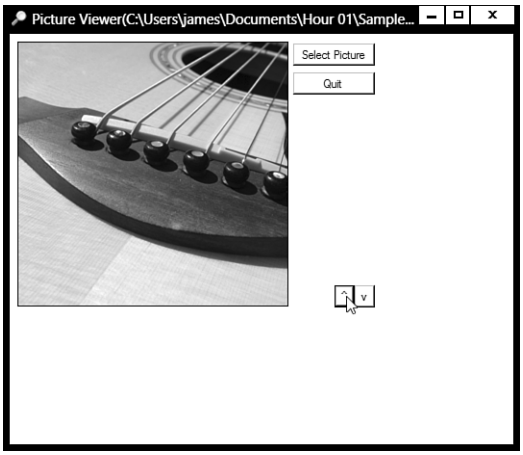

FIGURE 3.5
The form gets bigger, but it still looks just as you designed it.

4. 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 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, which describe the object. To understand this distinction, think about the pet object example one more time. A `Dog` object has a certain set of actions it can perform. These actions, called methods in Visual Basic, include barking, tail wagging, and chewing carpet (don't ask). Figure 3.6 illustrates the `Dog` object and its methods.

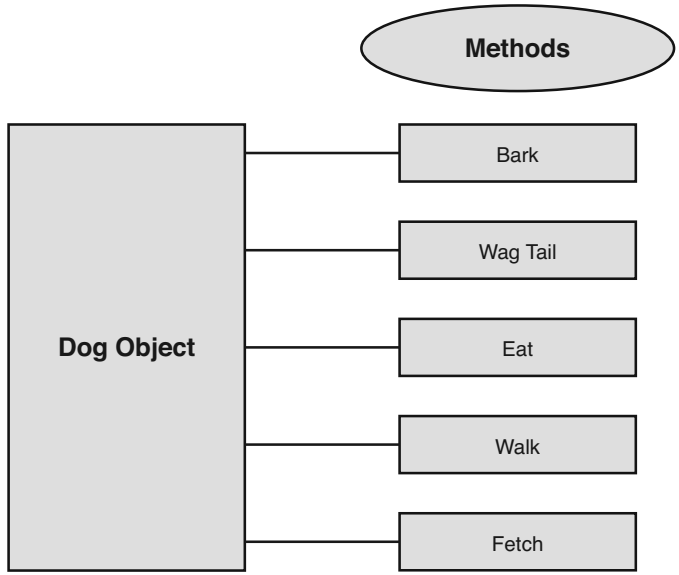

FIGURE 3.6
Invoking a method causes the object to perform an action.

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 not required to accept parameters (data passed to it by the calling code) or return a value; many methods simply perform an action in code. Invoking (triggering) a method is similar to referencing the value of a property. First you reference the object's name, and then provide a dot, and then the method name:

```
ObjectName.Method
```

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

```
Bruno.Bark()
```

By the Way

Methods generally are used to have an object perform an action, such as saving or deleting a record in a database. Properties, on the other hand, are used to get and set the object's attributes. One way to tell in code whether a statement is a property reference or method call is that a method call has a set of parentheses after it, like this:

```
AlbumForm.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 `0`, the `Walk()` and `Fetch()` methods obviously would be inapplicable. If you were to set the `NumberofLegs` property back to `4`, you could then trigger the `Walk()` or `Fetch()` methods again.

Building a Simple Object Example Project

The only way to really grasp what objects are and how they work is to use them. I've said this before, but I can't say it enough: Everything in Visual Basic 2012 is an object. This fact has its good points and bad points. One of the bad points is that in some instances, it takes more code to accomplish a task than it did in the past—sometimes more characters, sometimes more statements. If you're moving from Visual Basic 6, you have some learning and adjusting ahead of you, but it's worth the effort!

Every project you've built so far uses objects, but now you'll 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'll modify your Picture Viewer project to include a button that, when clicked, draws a colored border around the picture. You'll get a taste of some drawing functions in this example. Don't worry; you're not expected to understand all the intricacies of the drawing code. Your sole responsibility is grasping how objects work.

Creating the Interface for the Drawing Project

Continuing with the Picture Viewer project you've been using in this hour, add a new button to the form, and set its properties as follows:

Property	Value
Name	btnDrawBorder
Location	295,70
Size	85,23
Text	Draw Border

Writing the Object-Based Code

Now you'll add code to the button's `Click` event. I'll explain each statement, and at the end of the steps, I'll show the complete code listing. Follow these steps to create the code that draws the border:

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):

```
Dim objGraphics As Graphics
```

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, it creates the control objects), but not all objects are created automatically as they are in this situation. 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. The `Dim` statement you wrote in step 2 creates a new variable called `objGraphics`, which holds a reference to an object of type `Graphics`. You'll learn more about variables in Hour 11.

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

```
objGraphics = Me.CreateGraphics
```

`CreateGraphics` is a method of the form. (Remember, the keyword `Me` is shorthand for referencing the current form.) Under the hood, the `CreateGraphics` method is pretty complicated. 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 a form's borders and title bar. Anything drawn on the `objGraphics` object appears 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:

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

This statement clears the form's background 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 only to the right of a dot is a property reference or method call. If the text is followed by 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 are to look at the icon of the member in the IntelliSense drop-down or to consult the object's documentation.

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. You expect this string of object references to return a color value to be used to clear the graphic object. Therefore, you know that `Control` in this instance must be a property or 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. 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.

Enter the following statement. (Note: Do not press Enter until you're finished entering *all* the code shown here. The code appears on two lines only because of the size restriction of this page.)

```
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 equals sign is present but there is an open parenthesis, 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 won't go into detail on pens here; check the online help if pens interest you.) 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 pass property values. Specifically, you pass the top and left values for the picture, less 1. 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`. 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 1 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 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. However, the nature of using resources and the fact that you're responsible for telling your objects to clean up after themselves make 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—in an effort to smack you upside the head!

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:

```
Dim objGraphics As Graphics
objGraphics = Me.CreateGraphics
objGraphics.Clear(System.Drawing.SystemColors.Control)

objGraphics.DrawRectangle(System.Drawing.Pens.Blue, _
   picShowPicture.Left - 1, picShowPicture.Top - 1, _
   picShowPicture.Width + 1, picShowPicture.Height + 1)

objGraphics.Dispose()
```

By the Way

The statement calling `DrawRectangle()` is shown here as three lines of code. At the end of the first and second lines is an underscore character (_), also known as a *line continuation character*. It tells the Visual Basic compiler that the statement immediately following the character is a continuation of the current statement. You can, and should, use this character to break up long statements in your code.

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 as it did at design time. Clicking the button causes a blue rectangle to be drawn around the `PictureBox`, as shown in Figure 3.7.

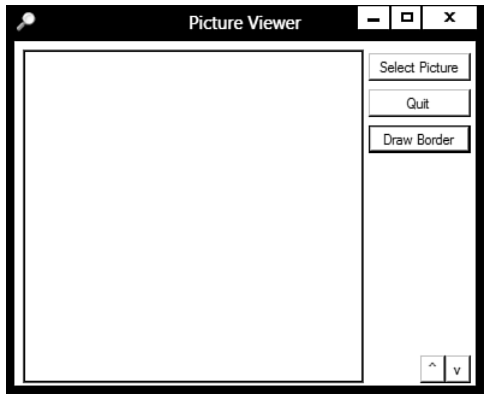

FIGURE 3.7
You can create simple lines and complex drawings by using objects.

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.

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 occurs any time you overlay the graphics with another form. In Hour 18, "Working with Graphics," you will learn how to persist images so that they don't disappear when the form becomes obscured.

Stop the project now by clicking Stop Debugging on the Visual Basic 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 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 requiring you to write the operative code for each object. In addition to containing an indexed set of objects, collections have properties and might have methods. Figure 3.8 illustrates the structure of a collection.

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 in 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.

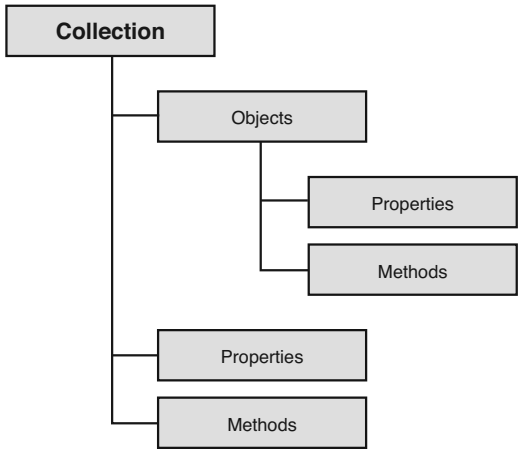

FIGURE 3.8
Collections contain sets of like objects, and they have their own properties and methods.

To better understand collections, you'll create a small Visual Basic 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 Basic (if it's not already loaded), and create a new Windows Application project titled **Collections Example**.
2. Change the form's filename to `CollectionsExampleForm.vb` using the Solution Explorer (right-click Form1.vb in the Solution Explorer and choose Rename), and set the form's Text property to **Collections Example** in the Properties window (you need to click the form first to view 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 `Label` 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:

```
Dim intIndex As Integer
For intIndex = 0 To Me.Controls.Count - 1
   MessageBox.Show("Control #" & intIndex & " has the name " & _
         Me.Controls(intIndex).Name)
Next intIndex
```

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 of the preceding code should look familiar to you by now. As with the `Object` example you created earlier, this statement creates a variable to hold a value. Rather than create a variable that can hold an object, as you did in the earlier example, this statement creates a variable that can hold only a number.

The next statement (the one that begins with `For`) accomplishes a few tasks. First, it initializes the variable `intIndex` to 0, and then it starts a loop (loops are discussed in Hour 14, "Looping for Efficiency"). It increments `intIndex` by 1 until `intIndex` equals the number of controls on the form, less 1. The reason you subtract 1 from the `Count` property is that collections are 0-based—the first item is always item 0. Thus, the first item is in index 0, the second item is in location 1, 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. You would be referencing an index that's 1 higher than the actual locations within the collection.

The `MessageBox.Show()` method (mentioned in Hour 2 and discussed in detail in Hour 17, "Interacting with Users") is a class of the .NET Framework that's used to display simple dialog boxes with text. The text that you're providing, which the `Show()` method displays, is a concatenation of multiple strings of text. (*Concatenation* is the process of adding strings together; it's 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 Debug, Stop Debugging to stop the program, and then save the project.

Because everything in Visual Basic 2012 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.

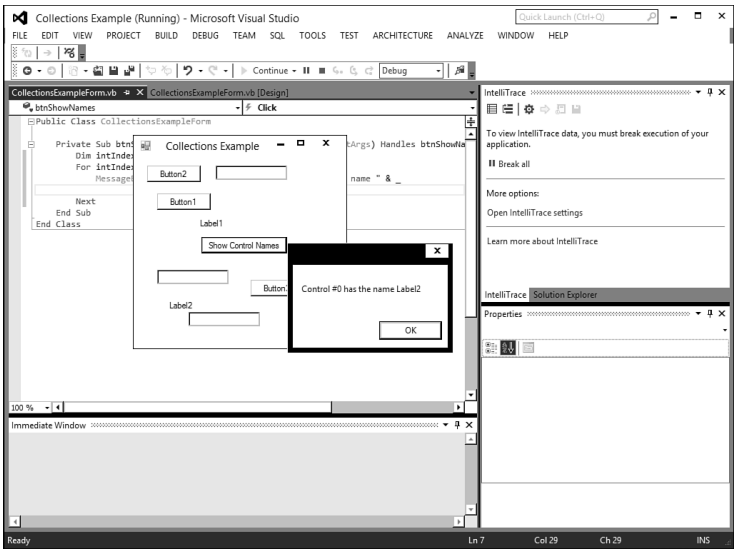

FIGURE 3.9
The Controls collection enables you to get to every control on a form.

Using the Object Browser

Visual Basic 2012 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 you're 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, press F2 or select it in the View menu.

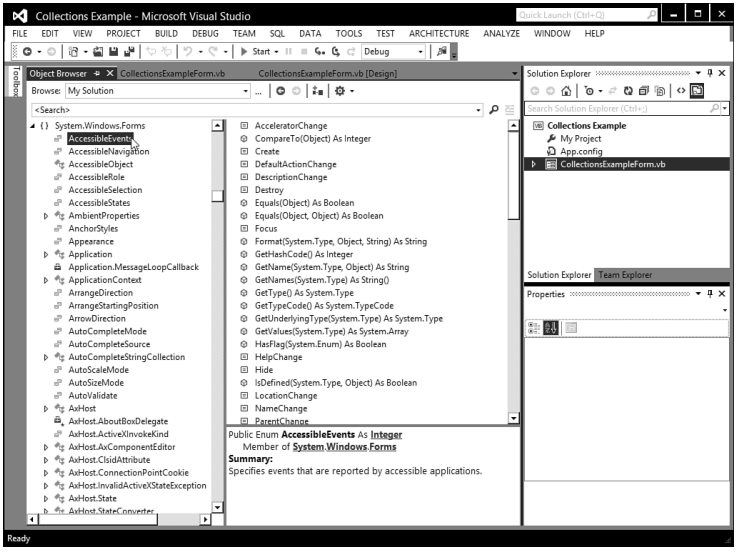

FIGURE 3.10
The Object Browser enables you to view all properties and methods of an object.

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 Basic .NET objects, as well as experience using the Object Browser.

The top-level nodes (each item in the tree is called 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 in 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 about 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. The Object Browser shows information about the member in the area below the member list.

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 in the Properties window, and most can also be set at runtime in Visual Basic code. You learned that referencing a property on the left side of the equals sign has the effect of changing the property, whereas referencing a property on the right side of the equals sign retrieves the 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 Basic, 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 with Visual Basic 2012.

Q&A

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

A. Absolutely. Visual Basic's context-sensitive Help extends to code as well as 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 `Integer` data type, you could type `Integer`, position the cursor within the word `Integer`, 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

Quiz

1. True or false: Visual Basic 2012 is a true object-oriented language.
2. An attribute that defines the state of an object is called a _____.
3. For you to change the value of a property, the property must be referenced on which side of the equals sign?
4. What is the term for when a new object is created from a template class?
5. An external function of an object (one that is available to code manipulating 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 `TextBox` controls.
2. Modify the collections example in this hour to print the height of all controls, rather than the name.

Index

B

C

D

E

F

G

H

I-J

K–L

M

N

O

P

Q–R

S

T

U

V

W

X–Y–Z