**Lucky Seven: Your First Visual Basic Program**

The Windows-based application you’re going to construct is Lucky Seven, a game program that simulates a lucky number slot machine . Lucky Seven has a simple user interface and can be created and compiled in just a few minutes using Microsoft Visual Basic . Here’s what your program will look like when it’s finished:



**37**

**Programming Steps**

The Lucky Seven user interface contains two buttons, three lucky number boxes, a digital photo depicting your winnings, and the label “Lucky Seven .” I produced these elements by creating seven objects on the Lucky Seven form and then changing several properties for each object . After I designed the interface, I added program code for the Spin and End buttons to process the user’s button clicks and produce the random numbers . To re-create Lucky Seven, you’ll follow three essential programming steps in Visual Basic: Create the user interface, set the properties, and write the program code . Table 2-1 shows the process for Lucky Seven .

**TABLE 2-1** **Building the Lucky Seven Program**

**Programming Step Number of Items**

1 . Create the user interface . 7 objects

|  |  |
| --- | --- |
| 2 . Set the properties . | 13 properties |
| 3 . Write the program code . | 2 objects |

**Creating the User Interface**

In this exercise, you’ll start building Lucky Seven by first creating a new project and then using controls in the Toolbox to construct the user interface .

**Create a new project**

1. Start Visual Studio 2010 .
2. On the Visual Studio File menu, click New Project .

**Tip**

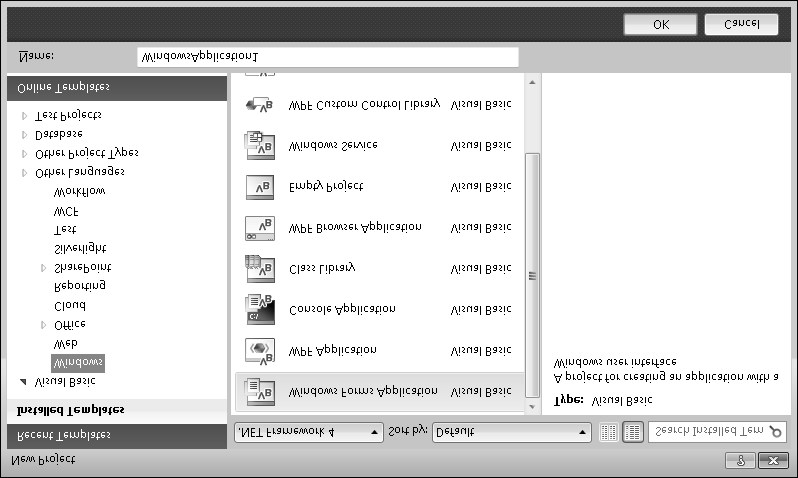
You can also start a new programming project by clicking the blue New Project link

on the Start Page .



The New Project dialog box opens, as shown on the following page .

The New Project dialog box provides access to the major project types available for writing Windows and Web applications . If you indicated during setup that you are a Visual Basic programmer, Visual Basic is your primary development option (as shown here), but the other languages in Visual Studio (Visual C#, Visual C++, and Visual F#) are always available through this dialog box . Although you will select a basic Windows



application project in this exercise, this dialog box is also the gateway to other types of development projects, such as a Web application, console application, Microsoft Office add-in, Windows Azure Cloud Service, Silverlight application, or Visual Studio deployment project .

Near the top of the New Project dialog box, you will notice a drop-down list box . This feature allows you to specify the version of the Microsoft .NET Framework that your application will target . This feature is sometimes called *multi-targeting*, meaning that through it, you can select the target environment that your program will run on . For example, if you retain the default selection of .NET Framework 4, any computer that your application will run on must have .NET Framework 4 installed . (Not to worry—the .NET Framework is usually installed as part of the operating system installation, or when you install a new Visual Basic program that you have written .) Unless you have a specific need, you can just leave this drop-down list at its default setting of .NET Framework 4 . Visual Basic 2010 Express does not include this drop-down list . You’ll learn more about the .NET Framework in Chapter 5, “Visual Basic Variables and Formulas, and the .NET Framework .”

1. Click the Windows Forms Application icon in the central Templates area of the dialog box, if it is not already selected .

Visual Studio prepares the development environment for Visual Basic Windows application programming .

1. In the Name text box, type **MyLucky7** .

Visual Studio assigns the name MyLucky7 to your project . (You’ll specify a folder location for the project later .) I’m recommending the “My” prefix here so you don’t confuse your new application with the Lucky7 project I’ve created for you on disk .

**Tip** If your New Project dialog box contains Location and Solution Name text boxes, you need to specify a folder location and solution name for your new programming project now . The presence of these text boxes is controlled by a check box in the Project And Solutions category of the Options dialog box, but it is not the default setting . (You display this dialog box by clicking the Options command on the Tools menu .) Throughout this book, you will be instructed to save your projects (or discard them) *after* you have completed the programming exercise . For more information about this “delayed saving” feature and default settings, see the section entitled “Customizing IDE Settings to Match Step-by-Step Exercises” in Chapter 1 .



1. Click OK to create the new project in Visual Studio .

Visual Studio cleans the slate for a new programming project and displays the blank Windows form that you will use to build your user interface .

Now you’ll enlarge the form and create the two buttons in the interface .

**Create the user interface**

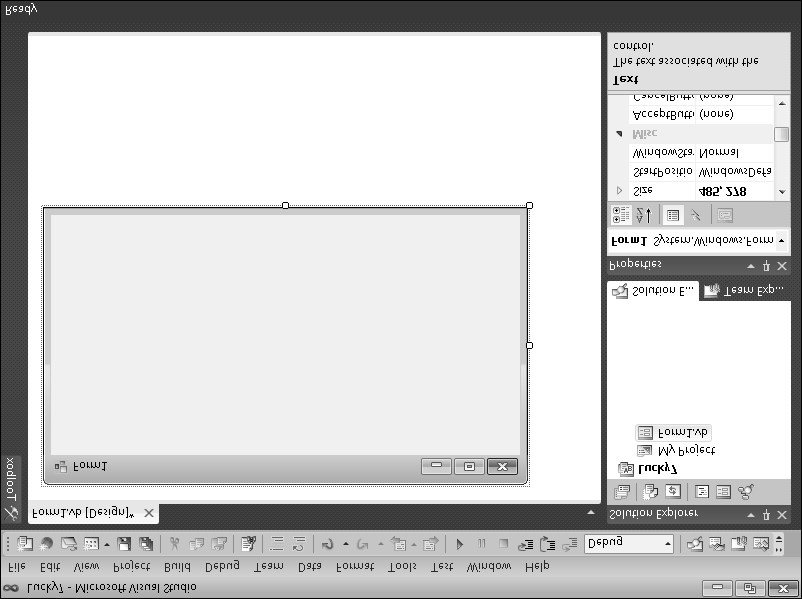
**1.** Point to the lower-right corner of the form until the mouse pointer changes to a r esizing pointer, and then drag to increase the size of the form to make room for the objects in your program .

As you resize the form, scroll bars might appear in the Designer to give you access to the entire form you’re creating . Depending on your screen resolution and the Visual Studio tools you have open, you might not be able to see the entire form at once . Don’t worry about this—your form can be small, or it can fill the entire screen because the scroll bars give you access to the entire form .

Size your form so that it is about the size of the form shown on the following page . If you want to match my example exactly, you can use the width and height dimensions (485 pixels × 278 pixels) shown in the lower-right corner of the screen .

To see the entire form without obstruction, you can resize or close the other programming tools, as you learned in Chapter 1 . (Return to Chapter 1 if you have questions about resizing windows or tools .)

Now you’ll practice adding a button object on the form .



1. Click the Toolbox tab to display the Toolbox window in the IDE .

The Toolbox contains all the controls that you’ll use to build Visual Basic programs in this book . The controls suitable for creating a Windows application are visible now because you selected the Windows Application project type earlier . Controls are organized by type, and by default the Common Controls category is visible . (If the Toolbox is not visible now, click Toolbox on the View menu to display it .)

1. Double-click the *Button* control in the Toolbox, and then move the mouse pointer away from the Toolbox .

Visual Studio creates a default-sized button object on the form and hides the Toolbox, as shown here:



The button is named *Button1* because it is the first button in the program . (You should make a mental note of this button name—you’ll see it again when you write your program code .) The new button object is selected and enclosed by resize handles . When Visual Basic is in *design mode* (that is, whenever the Visual Studio IDE is active), you can move objects on the form by dragging them with the mouse, and you can resize them by using the resize handles . While a program is running, however, the user can’t move user interface (UI) elements unless you’ve changed a property in the program to allow this . You’ll practice moving and resizing the button now .

**Move and resize a button**

1. Point to the button so that the pointer changes to a four-headed arrow, and then drag the button down and to the right .

The button moves across the surface of the form . If you move the object near the edge of the form or another object (if other objects are present), it automatically aligns itself to a hidden grid when it is an inch or so away . A little blue “snapline” also appears to help you gauge the distance of this object from the edge of the form or the other object . The grid is not displayed on the form by default, but you can use the snapline to judge distances with almost the same effect .

1. Position the mouse pointer on the lower-right corner of the button .

When the mouse pointer rests on a resize handle of a selected object, it becomes a resizing pointer . You can use the resizing pointer to change the size of an object .

1. Enlarge the button by dragging the pointer down and to the right .

When you release the mouse button, the button changes size and snaps to the grid .

1. Use the resizing pointer to return the button to its original size .

Now you’ll add a second button to the form, below the first button .

**Add a second button**

1. Click the Toolbox tab to display the Toolbox .
2. Click the *Button* control in the Toolbox (single-click this time), and then move the mouse pointer over the form .

The mouse pointer changes to crosshairs and a button icon . The crosshairs are designed to help you draw the rectangular shape of the button on the form, and you can use this method as an alternative to double-clicking to create a control of the default size .

1. Click and drag the pointer down and to the right . Release the mouse button to complete the button, and watch it snap to the form .
2. Resize the button object so that it is the same size as the first button, and then move it below the first button on the form . (Use the snapline feature to help you .)

**Tip** At any time, you can delete an object and start over again by selecting the object on the form and then pressing DELETE . Feel free to create and delete objects to practice creating your user interface .



Now you’ll add the labels used to display the numbers in the program . A *label* is a special user interface element designed to display text, numbers, or symbols when a program runs . When the user clicks the Lucky Seven program’s Spin button, three random numbers appear in the label boxes . If one of the numbers is a 7, the user wins .

**Add the number labels**

1. Double-click the *Label* control in the Toolbox .

Visual Studio creates a label object on the form . The label object is just large enough to hold the text contained in the object (it is rather small now), but it can be resized .

1. Drag the *Label1* object to the right of the two button objects .

Your form looks something like this:



1. Double-click the *Label* control in the Toolbox to create a second label object .

This label object will be named *Label2* in the program .

1. Double-click the *Label* control again to create a third label object .
2. Move the second and third label objects to the right of the first one on the form .

Allow plenty of space between the three labels because you will use them to display large numbers when the program runs .

Now you’ll use the *Label* control to add a descriptive label to your form . This will be the fourth and final label in the program .

1. Double-click the *Label* control in the Toolbox .
2. Drag the *Label4* object below the two command buttons .

When you’ve finished, your four labels should look like those in the following screen shot . (You can move your label objects if they don’t look quite right .)

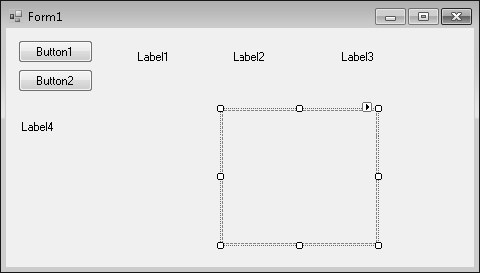


Now you’ll add a picture box to the form to graphically display the payout you’ll receive when you draw a 7 and hit the jackpot . A *picture box* is designed to display bitmaps, icons, digital photos, and other artwork in a program . One of the best uses for a picture box is to display a JPEG image file .

**Add a picture**

1. Click the *PictureBox* control in the Toolbox .
2. Using the control’s drawing pointer, create a large rectangular box below the second and third labels on the form .

Leave a little space below the labels for their size to grow as I mentioned earlier . When you’ve finished, your picture box object looks similar to this:



This object will be named *PictureBox1* in your program; you’ll use this name later in the program code .

Now you’re ready to customize your interface by setting a few properties .

**Setting the Properties**

As you discovered in Chapter 1, you can change properties by selecting objects on the form and changing their settings in the Properties window . You’ll start by changing the property settings for the two buttons .

**Set the button properties**

1. Click the first button (*Button1*) on the form .

The button is selected and is surrounded by resize handles .

1. Click the Properties window title bar .

**Tip**

If the Properties window isn’t visible, click the Properties Window command on the

View menu, or press F4 .



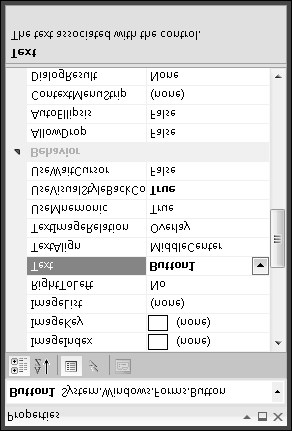
1. At the top of the Properties window, click the Categorized button .

For information about categorized properties, see the section entitled “The Properties Window” in Chapter 1 .

1. Resize the Properties window (if necessary) so that there is plenty of room to see the property names and their current settings .

Once you get used to setting properties, you will probably use the Properties w indow without enlarging it, but making it bigger helps when you first try to use it . The

Properties window in the following screen shot is a good size for setting properties:



The Properties window lists the settings for the first button . These include settings for the background color, text, font height, and width of the button . Because there are so many properties, Visual Studio organizes them into categories and displays them in outline view . If you want to see the properties in a category, click the arrow sign (>) next to the category title .

1. If it is not already visible, scroll in the Properties window until you see the *Text* property located in the Appearance category .
2. Double-click the *Text* property in the first column of the Properties window .

The current *Text* setting (“Button1”) is highlighted in the Properties window .

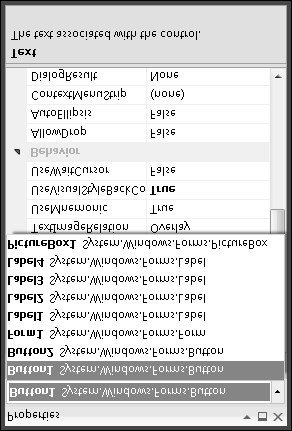
1. Type **Spin**, and then press ENTER .

The *Text* property changes to “Spin” in the Properties window and on the button on the form . Now you’ll change the *Text* property of the second button to “End .”

(You’ll select the second button in a new way this time .)

1. Open the Object list at the top of the Properties window .

A list of the interface objects in your program appears as follows:



1. Click Button2 System .Windows .Forms .Button (the second button) in the list box .

The property settings for the second button appear in the Properties window, and Visual Studio highlights Button2 on the form .

1. Double-click the current *Text* property (“Button2”), type **End**, and then press ENTER .

The text of the second button changes to “End .”

**Tip** Using the Object list is a handy way to switch between objects in your program .



You can also switch between objects on the form by clicking each object .

Now you’ll set the properties for the labels in the program . The first three labels will hold the random numbers generated by the program and will have identical property settings . (You’ll set most of them as a group .) The descriptive label settings will be slightly different .

**Set the number label properties**

1. Click the first number label (*Label1*), hold down the SHIFT key, click the second and third number labels, and then release the SHIFT key . (If the Properties window is in the way, move it to a new place .)
   1. selection rectangle and resize handles appear around each label you click . You’ll change the *TextAlign*, *BorderStyle*, and *Font* properties now so that the numbers that will appear in the labels will be centered, boxed, and identical in font and font size . (All these properties are located in the Appearance category of the Properties window .) You’ll also set the *AutoSize* property to False so that you can change the size of the labels according to your precise specifications . (The *AutoSize* property is located in the Layout category .)

**Tip** When more than one object is selected, only those properties that can be changed for the group are displayed in the Properties window .



1. Click the *AutoSize* property in the Properties window, and then click the arrow that appears in the second column .
2. Set the *AutoSize* property to False so that you can size the labels manually .
3. Click the *TextAlign* property, and then click the arrow that appears in the second column .
   1. graphical assortment of alignment options appears in the list box; you can use these settings to align text anywhere within the borders of the label object .
4. Click the center option (MiddleCenter) .

The *TextAlign* property for each of the selected labels changes to MiddleCenter .

1. Click the *BorderStyle* property, and then click the arrow that appears in the second column .

The valid property settings (None, FixedSingle, and Fixed3D) appear in the list box .

1. Click FixedSingle in the list box to add a thin border around each label .
2. Click the *Font* property, and then click the ellipsis button (the button with three dots that’s located next to the current font setting) .

The Font dialog box opens .

1. Change the font to Times New Roman, the font style to Bold, and the font size to 24, and then click OK .

The label text appears in the font, style, and size you specified .

Now you’ll set the text for the three labels to the number 0—a good “placeholder” for the numbers that will eventually fill these boxes in your game . (Because the program produces the actual numbers, you could also delete the text, but putting a placeholder here gives you something to base the size of the labels on .)

1. Click a blank area on the form to remove the selection from the three labels, and then

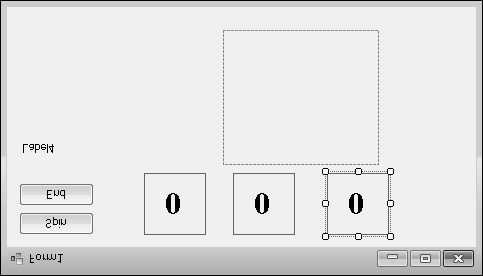
click the first label .

1. Double-click the *Text* property, type **0**, and then press ENTER .

The text of the *Label1* object is set to 0 . You’ll use program code to set this property to a random “slot machine” number later in this chapter .

1. Change the text in the second and third labels on the form to **0** also .
2. Move and resize the labels now so that they are appropriately spaced .

Your form looks something like this:



Now you’ll change the *Text*, *Font*, and *ForeColor* properties of the fourth label .

**Set the descriptive label properties**

1. Click the fourth label object (*Label4*) on the form .
2. Change the *Text* property in the Properties window to **Lucky Seven** .
3. Click the *Font* property, and then click the ellipsis button .
4. Use the Font dialog box to change the font to Arial, the font style to Bold, and the font size to 18 . Then click OK .

The font in the *Label4* object is updated, and the label is resized automatically to hold the larger font size because the object’s *AutoSize* property is set to True .

1. Click the *ForeColor* property in the Properties window, and then click the arrow in the second column .

Visual Studio displays a list box with Custom, Web, and System tabs for setting the foreground colors (the color of text) of the label object . The Custom tab offers many of the colors available in your system . The Web tab sets colors for Web pages and lets you pick colors using their common names . The System tab displays the current colors used for user interface elements in your system .

1. Click the purple color on the Custom tab .

The text in the label box changes to purple .

Now you’re ready to set the properties for the last object .

**The Picture Box Properties**

When the person playing your game hits the jackpot (that is, when at least one 7 appears in the number labels on the form), the picture box object will contain a picture in JPEG format of a person dispensing money . (I am supplying you with this digitized image, but you can substitute your own if you like .) You need to set the *SizeMode* property to accurately size the picture and set the *Image* property to specify the name of the JPEG file that you will load into the picture box . You also need to set the *Visible* property, which specifies the picture state at the beginning of the program .

**Set the picture box properties**

1. Click the picture box object on the form .
2. Click the *SizeMode* property in the Properties window (listed in the Behavior category), click the arrow in the second column, and then click StretchImage .

Setting *SizeMode* to StretchImage before you open a graphic causes Visual Studio to resize the graphic to the exact dimensions of the picture box . (Typically, you set this property before you set the *Image* property .)

1. Click the *Image* property in the Properties window, and then click the ellipsis button in the second column .

The Select Resource dialog box opens .

1. Click the Local Resource radio button, and then click the Import button .
2. In the Open dialog box, navigate to the C:\Vb10sbs\Chap02 folder .

This folder contains the digital photo PayCoins .jpg .

1. Select PayCoins .jpg, and then click Open .

An screen shot of one person paying another appears in the Select Resource dialog box . (The letter “W” represents winning .)



1. Click OK .

The PayCoins photo is loaded into the picture box . Because the photo is relatively small (24 KB), it opens quickly on the form .

1. Resize the picture box object now to fix any distortion problems that you see in the image .

I sized my picture box object to be 144 pixels wide by 146 pixels high . You can match this size by using the width and height dimensions located on the lower-right side of the Visual Studio IDE . (The dimensions of the selected object are given on the lower-right side, and the location on the form of the object’s upper-left corner is given to the left of the dimensions .)

This particular image displays best when the picture box object retains a square shape .

**Note** As you look at the picture box object, you might notice a tiny shortcut arrow called a *smart tag* near its upper-right corner . This smart tag is a button that you can click to quickly change a few common picture box settings and open the Select Resource dialog box . (You’ll see the smart tag again in Chapter 4, “Working with Menus, Toolbars, and Dialog Boxes,” when you use the *ToolStrip* control .)



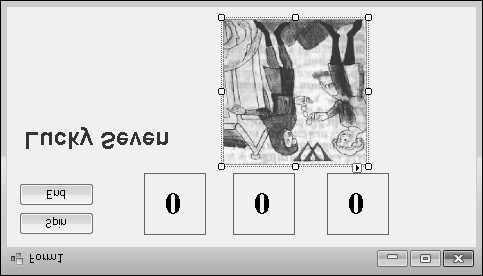
Now you’ll change the *Visible* property to False so that the image will be invisible when the program starts .

1. Click the *Visible* property in the Behavior category of the Properties window, and then click the arrow in the second column .

The valid settings for the *Visible* property appear in a list box .

1. Click False to make the picture invisible when the program starts .

Setting the *Visible* property to False affects the picture box when the program runs, but not now, while you’re designing it . Your completed form looks similar to this:



**Tip**

You can also double-click property names that have True and False settings (so-called

Boolean properties), to toggle back and forth between True and False . Default Boolean

properties are shown in regular type, and changed settings appear in bold .



1. You are finished setting properties for now, so if your Properties window is fl oating, hold down the CTRL key and double-click its title bar to return it to the docked position .

**Reading Properties in Tables**

In this chapter, you’ve set the properties for the Lucky Seven program step by step . In future chapters, the instructions to set properties will be presented in table format unless a setting is especially tricky . Table 2-2 lists the properties you’ve set so far in the Lucky Seven program, as they’d look later in the book . Settings you need to type in are shown in quotation marks . You shouldn’t type the quotation marks .

**TABLE 2-2 Lucky Seven Properties**

**Object Property Setting**

*Button1 Text* “Spin”

*Button2 Text* “End”

*Label1, Label2, Label3 AutoSize* False

*BorderStyle* FixedSingle

*Font* Times New Roman, Bold, 24-point

*Text* “0”

*TextAlign* MiddleCenter

*Label4 Text* “Lucky Seven”

*Font* Arial, Bold, 18-point

*ForeColor* Purple

*PictureBox1 Image* “C:\Vb10sbs\Chap02\Paycoins .jpg”

*SizeMode* StretchImage

*Visible* False

**Writing the Code**

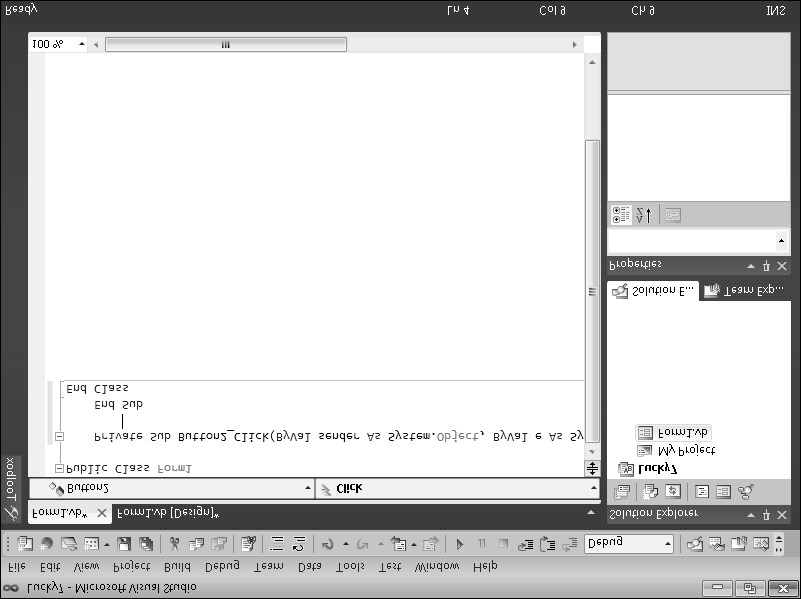
Now you’re ready to write the code for the Lucky Seven program . Because most of the objects you’ve created already “know” how to work when the program runs, they’re ready to receive input from the user and process it . The inherent functionality of objects is one of the great strengths of Visual Studio and Visual Basic—after objects are placed on a form and their properties are set, they’re ready to run without any additional programming . However, the “meat” of the Lucky Seven game—the code that actually calculates random numbers, displays them in boxes, and detects a jackpot—is still missing from the program . This computing logic can be built into the application only by using program statements— code that clearly spells out what the program should do at each step of the way . Because the Spin and End buttons drive the program, you’ll associate the code for the game with those buttons . You enter and edit Visual Basic program statements in the Code Editor .

In the following steps, you’ll enter the program code for Lucky Seven in the Code Editor .

**Use the Code Editor**

**1.** Double-click the End button on the form .

The Code Editor appears as a tabbed document window in the center of the Visual Studio IDE, as shown here:



Inside the Code Editor are program statements associated with the current form . Program statements that are used together to perform some action are typically grouped in a programming construct called a *procedure* . A common type of procedure is a Sub procedure, sometimes called a *subroutine* . Sub procedures include a *Sub* keyword in the first line and end with  *End Sub* . (I’ll talk about the Public and Private keywords later .) Procedures are typically executed when certain events occur, such as when a button is clicked . When a procedure is associated with a particular object and an event, it is called an *event handler* or an *event procedure* .

When you double-clicked the End button (*Button2*), Visual Studio automatically added the first and last lines of the *Button2\_Click* event procedure, as the following code shows . (The first line was wrapped to stay within the book margins .) You may notice other bits of code in the Code Editor (words like *Public* and *Class*), which Visual Studio has added to define important characteristics of the form, but I won’t emphasize them here .

Private Sub Button2\_Click(ByVal sender As System.Object, \_

ByVal e As System.EventArgs) Handles Button2.Click

End Sub

The body of a procedure fits between these lines and is executed whenever a user activates the interface element associated with the procedure . In this case, the event is a mouse click, but as you’ll see later in the book, it could also be a different type of event .

**2.** Type **End**, and then press the ENTER key .

When you type the statement, Visual Studio recognizes *End* as a unique reserved word or *keyword* and displays it in a list box with Common and All tabs . Microsoft calls this auto-extend feature IntelliSense because it tries to intelligently help you write code, and you can browse through various Visual Basic keywords and objects alphabetically . (In this way, the language is partially discoverable through the IDE itself .)

After you press the ENTER key, the letters in *End* turn blue and are indented, indicating that Visual Basic recognizes *End* as one of several hundred unique keywords within the Visual Basic language . You use the *End* keyword to stop your program and remove it from the screen . In this case, *End* is also a complete *program statement*, a s elf-contained instruction executed by the *Visual Basic compiler*, the part of Visual Studio that processes or *parses* each line of Visual Basic *source code*, combining the result with other resources to create an executable file . Program statements are a little like c omplete sentences in a human language—statements can be of varying lengths but must follow the grammatical “rules” of the compiler . In Visual Studio, program statements can be composed of keywords, properties, object names, variables, numbers, special symbols, and other values . You’ll learn more about how program statements are constructed in Chapter 5 .

As you enter program statements and make other edits, the Code Editor handles many of the formatting details for you, including adjusting indentation and spacing and adding any necessary parentheses . The exact spelling, order, and spacing of items within program statements is referred to as *statement syntax* . In the early days of compilers, programmers were almost totally responsible for getting the precise syntax for each program statement correct on their own, but now sophisticated development tools such as Visual Studio help immensely with the construction of accurate program statements .

When you pressed the ENTER key, the *End* statement was indented to set it apart from the *Private Sub* and *End Sub* statements . This indenting scheme is one of the programming conventions you’ll see throughout this book to keep your programs clear and readable . The group of conventions regarding how code is organized in a program is often referred to as *program style* .

Now that you’ve written the code associated with the End button, you’ll write code for the Spin button . These program statements will be a little more extensive and will give you a chance to learn more about statement syntax and program style . You’ll study many of the program statements later in this book, so you don’t need to know everything about them now . Just focus on the general structure of the code and on typing the program statements exactly as they are printed .

**Write code for the Spin button**

1. At the top of the Solution Explorer window, click the View Designer button in the Solution Explorer window to display your form again .

**Note** When the Code Editor is visible, you won’t be able to see the form you’re working on . The View Designer button is one mechanism you can use to display it again . (If more than one form is loaded in Solution Explorer, click the form that you want to display first .) You can also click the Form1 .vb [Design] tab at the top edge of the Code Editor . To display the Code Editor again, click the View Code button in Solution Explorer .



1. Double-click the Spin button .

After a few moments, the Code Editor appears, and an event procedure associated with the *Button1* button appears near the *Button2* event procedure .

Although you changed the text of this button to “Spin,” its name in the program is still *Button1* . (The name and the text of an interface element can be different to suit the needs of the programmer .) Each object can have several procedures associated with it, one for each event it recognizes . The click event is the one you’re interested in now because users will click the Spin and End buttons when they run the program .

1. Type the following program lines between the *Private Sub* and *End Sub* statements . Press ENTER after each line, press TAB to indent, and take care to type the program statements exactly as they appear here . (The Code Editor will scroll to the left as you enter the longer lines .) If you make a mistake (usually identified by a jagged underline), delete the incorrect statements and try again .

**Tip** As you enter the program code, Visual Basic formats the text and displays different parts of the program in color to help you identify the various elements . When you begin to type a property, Visual Basic also displays the available properties for the object you’re using in a list box, so you can double-click the property or keep typing to enter it yourself . If Visual Basic displays an error message, you might have misspelled a program statement . Check the line against the text in this book, make the necessary correction, and continue typing . (You can also delete a line and type it from scratch .) In addition, Visual Basic might add necessary code automatically . For example, when you type the following code, Visual Basic automatically adds the *End If* line . Readers of previous editions of this book have found this first typing exercise to be the toughest part of this chapter—“But Mr . Halvorson, I know I typed it just as you wrote it!”—so please give this program code your closest attention . I promise you, it works!



**PictureBox1.Visible = False ' hide picture Label1.Text = CStr(Int(Rnd() \* 10)) ' pick numbers**

**Label2.Text = CStr(Int(Rnd() \* 10))**

**Label3.Text = CStr(Int(Rnd() \* 10))**

**' if any number is 7 display picture and beep**

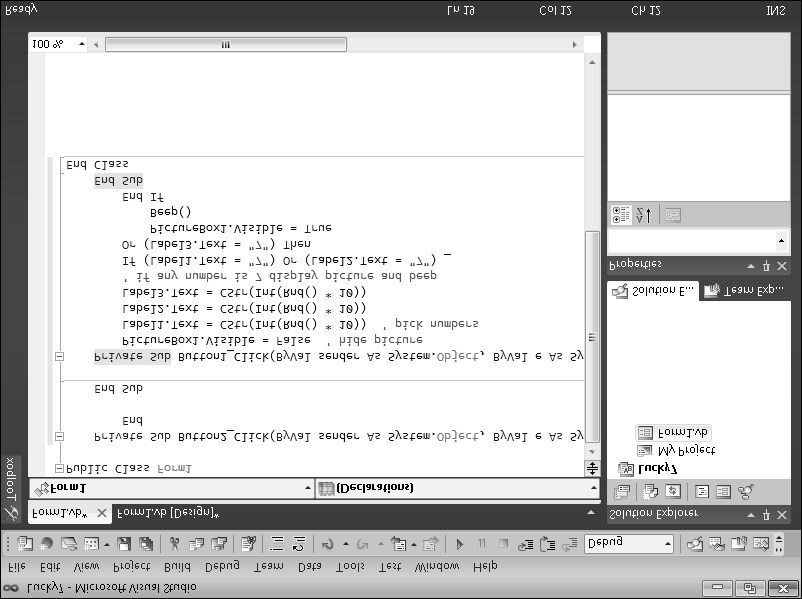
**If (Label1.Text = "7") Or (Label2.Text = "7") \_**

**Or (Label3.Text = "7") Then PictureBox1.Visible = True**

**Beep()**

**End If**

When you’ve finished, the Code Editor looks as shown in the following screen shot:



1. Click the Save All command on the File menu to save your additions to the program .

The Save All command saves everything in your project—the project file, the form file, any code modules, and other related components in your application . Since this is the first time that you have saved your project, the Save Project dialog box opens, prompting you for the name and location of the project . (If your copy of Visual Studio is configured to prompt you for a location when you first create your project, you won’t see the Save Project dialog box now—Visual Studio just saves your changes .)

1. Browse and select a location for your files .

I recommend that you use the C:\Vb10sbs\Chap02 folder (the location of the book’s sample files), but the location is up to you . Since you used the “My” prefix when you originally opened your project, this version won’t overwrite the Lucky7 practice file that I built for you on disk .

1. Clear the Create Directory For Solution check box .

When this check box is selected, it creates a second folder for your program’s solution files, which is not necessary for solutions that contain only one project (the situation for most programs in this book) .

1. Click Save to save your files .

**Note** If you want to save just the item you are currently working on (the form, the code module, or something else), you can use the Save command on the File menu . If you want to save the current item with a different name, you can use the Save As command .



**A Look at the *Button1\_Click* Procedure**

The *Button1\_Click* procedure is executed when the user clicks the Spin button on the form . The procedure uses some pretty complicated statements, and because I haven’t formally introduced them yet, it might look a little confusing . However, if you take a closer look, you’ll probably see a few things that look familiar . Taking a peek at the contents of these procedures will give you a feel for the type of program code you’ll be creating later in this book . (If you’d rather not stop for this preview, feel free to skip to the next section, “Running Visual Basic Applications .”)

The *Button1\_Click* procedure performs three tasks:

n It hides the digital photo . n It creates three random numbers for the number labels . n It displays the photo when the number 7 appears .

Let’s look at each of these steps individually .

Hiding the photo is accomplished with the following line:

PictureBox1.Visible = False ' hide picture

This line is made up of two parts: a program statement and a comment .

The *PictureBox1.Visible = False* program statement sets the *Visible* property of the picture box object (*PictureBox1*) to False (one of two possible settings) . You might remember that you set this property to False once before by using the Properties window . You’re d oing it again now in the program code because the first task is a spin and you need to clear away a photo that might have been displayed in a previous game . Because the property will be changed at run time and not at design time, you must set the property by using program code . This is a handy feature of Visual Basic, and I’ll talk about it more in Chapter 3, “Working with Toolbox Controls .”

The second part of the first line (the part displayed in green type on your screen) is called a *comment* . Comments are explanatory notes included in program code following a single quotation mark (‘) . Programmers use comments to describe how important statements work in a program . These notes aren’t processed by Visual Basic when the program runs; they exist only to document what the program does . You’ll want to use comments often when you write Visual Basic programs to leave an easy-to-understand record of what you’re doing .

The next three lines handle the random number computations . Does this concept sound strange? You can actually make Visual Basic generate unpredictable numbers within specific guidelines—in other words, you can create random numbers for lottery contests, dice games, or other statistical patterns . The *Rnd* function in each line creates a random number between 0 and 1 (a number with a decimal point and several decimal places), and the *Int* function returns the integer portion of the result of multiplying the random number by 10 . This computation creates random numbers between 0 and 9 in the program—just what you need for this particular slot machine application .

Label1.Text = CStr(Int(Rnd() \* 10)) ' pick numbers

You then need to jump through a little hoop in your code . You need to copy these random numbers into the three label boxes on the form, but first the numbers need to be c onverted to text with the *CStr* (convert to string) function . Notice how *CStr*, *Int*, and *Rnd* are all connected in the program statement—they work collectively to produce a result like a m athematical formula . After the computation and conversion, the values are assigned to the *Text* properties of the first three labels on the form, and the assignment causes the numbers to be displayed in bold, 24-point, Times New Roman font in the three number labels .

The last group of statements in the program checks whether any of the random numbers is 7 . If one or more of them is, the program displays the graphical depiction of a payout, and a beep announces the winnings .

' if any number is 7 display picture and beep

If (Label1.Text = "7") Or (Label2.Text = "7") \_

Or (Label3.Text = "7") Then PictureBox1.Visible = True

Beep()

End If

Each time the user clicks the Spin button, the *Button1\_Click* procedure is executed, or *called*, and the program statements in the procedure are run again .

**Running Visual Basic Applications**

Congratulations! You’re ready to run your first real program . To run a Visual Basic program from the development environment, you can do any of the following: n Click Start Debugging on the Debug menu . n Click the Start Debugging button on the Standard toolbar .

n Press F5 .

Try running your Lucky Seven program now . If Visual Basic displays an error message, you might have a typing mistake or two in your program code . Try to fix it by comparing the printed version in this book with the one you typed, or load Lucky7 from your hard disk and run it .

**Run the Lucky Seven program**

1. Click the Start Debugging button on the Standard toolbar .

The Lucky Seven program compiles and runs in the IDE . After a few seconds, the user interface appears, just as you designed it .

1. Click the Spin button .

The program picks three random numbers and displays them in the labels on the form, as follows:



Because a 7 appears in the first label box, the digital photo depicting the payoff appears, and the computer beeps . You win! (The sound you hear depends on your Default Beep setting in the Sound Control Panel . To make this game sound really cool, change the Default Beep sound to something more dynamic .)

1. Click the Spin button 15 or 16 more times, watching the results of the spins in the number boxes .

About half the time you spin, you hit the jackpot—pretty easy odds . (The actual odds are about 2 .8 times out of 10; you’re just lucky at first .) Later on, you might want to make the game tougher by displaying the photo only when two or three 7s appear, or by creating a running total of winnings .

1. When you’ve finished experimenting with your new creation, click the End button .

The program stops, and the development environment reappears on your screen .

**Tip** If you run this program again, you might notice that Lucky Seven displays exactly the same sequence of random numbers . There is nothing wrong here—the Visual Basic *Rnd* function was designed to display a *repeating* sequence of numbers at first so that you can properly test your code using output that can be reproduced again and again . To c reate truly “random” numbers, use the *Randomize* function in your code, as shown in the exercise at the end of this chapter . The .NET Framework, which you’ll learn to use later, also supplies random number

