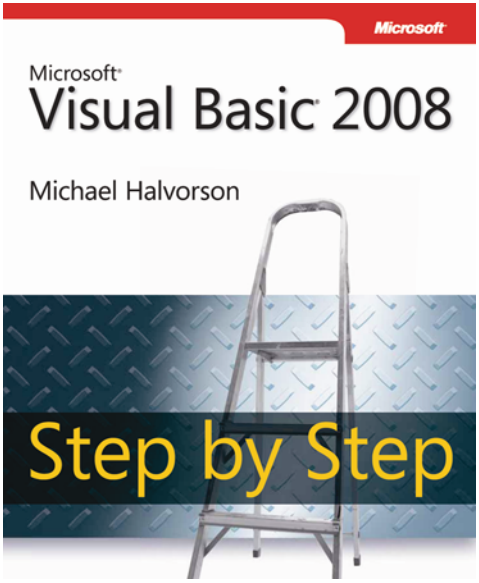


Microsoft® Visual Basic® 2008 Step by Step

Michael Halvorson

To learn more about this book, visit Microsoft Learning at
http://www.microsoft.com/MSPress/books/12202.aspx

9780735625372

Microsoft® Press

Table of Contents

Part IV Database and Web Programming

Appendix

What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Chapter 5

Visual Basic Variables and Formulas, and the .NET Framework

After completing this chapter, you will be able to:

- Use variables to store data in your programs.
- Get input by using the *InputBox* function.
- Display messages by using the *MsgBox* function.
- Work with different data types.
- Use variables and operators to manipulate data.
- Use methods in the .NET Framework.
- Use arithmetic operators and functions in formulas.

In this chapter, you'll learn how to use variables and constants to store data temporarily in your program, and how to use the *InputBox* and *MsgBox* functions to gather and present information by using dialog boxes. You'll also learn how to use functions and formulas to perform calculations, and how to use arithmetic operators to perform tasks such as multiplication and string concatenation. Finally, you'll learn how to tap into the powerful classes and methods of the Microsoft .NET Framework 3.5 to perform mathematical calculations and other useful work.

The Anatomy of a Visual Basic Program Statement

As you learned in Chapter 2, "Writing Your First Program," a line of code in a Visual Basic program is called a *program statement*. A program statement is any combination of Visual Basic keywords, properties, object names, variables, numbers, special symbols, and other values that collectively create a valid instruction recognized by the Visual Basic compiler. A complete program statement can be a simple keyword, such as

```
End
```

which halts the execution of a Visual Basic program, or it can be a combination of elements, such as the following statement, which uses the *TimeString* property to assign the current system time to the *Text* property of the *Label1* object:

```
Label1.Text = TimeString
```

The rules of construction that must be used when you build a programming statement are called *statement syntax*. Visual Basic shares many of its syntax rules with earlier versions of the BASIC programming language and with other language compilers. The trick to writing good program statements is learning the syntax of the most useful language elements and then using those elements correctly to process the data in your program. Fortunately, Visual Basic does a lot of the toughest work for you, so the time you spend writing program code is relatively short, and you can reuse the results in future programs. The Visual Studio IDE also points out potential syntax errors and suggests corrections, much like the AutoCorrect feature of Microsoft Office Word.

In this chapter and the following chapters, you'll learn the most important Visual Basic keywords and program statements, as well as many of the objects, properties, and methods provided by Visual Studio controls and the .NET Framework. You'll find that these keywords and objects complement nicely the programming skills you've already learned and will help you write powerful programs in the future. The first topics—variables and data types—are critical features of nearly every program.

Using Variables to Store Information

A *variable* is a temporary storage location for data in your program. You can use one or many variables in your code, and they can contain words, numbers, dates, properties, or other values. By using variables, you can assign a short and easy-to-remember name to each piece of data you plan to work with. Variables can hold information entered by the user at run time, the result of a specific calculation, or a piece of data you want to display on your form. In short, variables are handy containers that you can use to store and track almost any type of information.

Using variables in a Visual Basic program requires some planning. Before you can use a variable, you must set aside memory in the computer for the variable's use. This process is a little like reserving a seat at a theater or a baseball game. I'll cover the process of making reservations for, or *declaring*, a variable in the next section.

Setting Aside Space for Variables: The *Dim* Statement

Since the release of Microsoft Visual Basic .NET 2003, it has been necessary for Visual Basic programmers to explicitly declare variables before using them. This was a change from Visual Basic 6 and earlier versions of Visual Basic, where (under certain circumstances) you could declare variables implicitly—in other words, simply by using them and without a *Dim* statement. The earlier practice was flexible but rather risky—it created the potential for variable confusion and misspelled variable names, which introduced potential bugs into the code that might or might not be discovered later.

In Visual Basic 2008, a bit of the past has returned in the area of variable declaration. It is possible once again to declare a variable implicitly. I don't recommend this, however, so I won't discuss this new feature until you learn the recommended programming practice, which experienced programmers far and wide will praise you for adopting.

To declare a variable in Visual Basic 2008, type the variable name after the *Dim* statement. (Dim stands for *dimension*.) This declaration reserves room in memory for the variable when the program runs and lets Visual Basic know what type of data it should expect to see later. Although this declaration can be done at any place in the program code (as long as the declaration happens before the variable is used), most programmers declare variables in one place at the top of their event procedures or code modules.

For example, the following statement creates space for a variable named *LastName* that will hold a textual, or *string*, value:

```
Dim LastName As String
```

Note that in addition to identifying the variable by name, I've used the *As* keyword to give the variable a particular type, and I've identified the type by using the keyword *String*. (You'll learn about other data types later in this chapter.) A string variable contains textual information: words, letters, symbols—even numbers. I find myself using string variables a lot; they hold names, places, lines from a poem, the contents of a file, and many other "wordy" data.

Why do you need to declare variables? Visual Basic wants you to identify the name and the type of your variables in advance so that the compiler can set aside the memory the program will need to store and process the information held in the variables. Memory management might not seem like a big deal to you (after all, modern personal computers have lots of RAM and gigabytes of free hard disk space), but in some programs, memory can be consumed quickly, and it's a good practice to take memory allocation seriously even as you take your first steps as a programmer. As you'll soon see, different types of variables have different space requirements and size limitations.

Note In some earlier versions of Visual Basic, specific variable types (such as *String* or *Integer*) aren't required—information is simply held by using a generic (and memory hungry) data type called *Variant*, which can hold data of any size or format. Variants are not supported in Visual Basic 2008. Although they are handy for beginning programmers, their design makes them slow and inefficient, and they allow variables to be converted from one type to another too easily—often causing unexpected results. As you'll learn later, however, you can still store information in generic containers called *Object*, which are likewise general-purpose in function but rather inefficient in size.

After you declare a variable, you're free to assign information to it in your code by using the assignment operator (=). For example, the following program statement assigns the last name "Jefferson" to the *LastName* variable:

```
LastName = "Jefferson"
```

Note that I was careful to assign a textual value to the *LastName* variable because its data type is *String*. I can also assign values with spaces, symbols, or numbers to the variable, such as

```
LastName = "1313 Mockingbird Lane"
```

but the variable is still considered a string value. The number portion could be used in a mathematical formula only if it were first converted to an integer or a floating-point value by using one of a handful of conversion functions I'll discuss later in this book.

After the *LastName* variable is assigned a value, it can be used in place of the name "Jefferson" in your code. For example, the assignment statement

```
Label1.Text = LastName
```

displays "Jefferson" in the label named *Label1* on your form.

Implicit Variable Declaration

If you really want to declare variables "the old way" in Visual Basic 2008—that is, without explicitly declaring them by using the *Dim* statement—you can place the *Option Explicit Off* statement at the very top of your form's or module's program code (before any event procedures), and it will turn off the Visual Basic default requirement that variables be declared before they're used. As I mentioned earlier, I don't recommend this statement as a permanent addition to your code, but you might find it useful temporarily as you convert older Visual Basic programs to Visual Studio 2008.

Another possibility is to use the new *Option Infer* statement, which has been added to Visual Basic 2008. If *Option Infer* is set to "On", Visual Basic will deduce or *infer* the type of a variable by examining the initial assignment you make. This allows you to declare variables without specifically identifying the type used, and allowing Visual Basic to make the determination. For example, the expression

```
Dim attendance = 100
```

will declare the variable named *attendance* as an *Integer*, because 100 is an integer expression. In other words, with *Option Infer* set to "On", it is the same as typing

```
Dim attendance As Integer = 100
```

Likewise, the expression

```
Dim address = "1012 Daisy Lane"
```

will declare the variable address as type *String*, because its initial assignment was of type *String*. If you set *Option Infer* to "Off", however, Visual Basic will declare the variable as type *Object*—a general (though somewhat bulky and inefficient) container for any type of data.

If you plan to use *Option Infer* to allow this type of inferred variable declaration (a flexible approach, but one that could potentially lead to unexpected results), place the following two statements at the top of your code module (above the *Class Form* statement):

```
Option Explicit Off
Option Infer On
```

Option Explicit Off allows variables to be declared as they are used, and *Option Infer On* allows Visual Basic to determine the type automatically. You can also set these options using the Options command on the Tools menu as discussed in Chapter 1, "Exploring the Visual Studio Integrated Development Environment."

Using Variables in a Program

Variables can maintain the same value throughout a program, or they can change values several times, depending on your needs. The following exercise demonstrates how a variable named *LastName* can contain different text values and how the variable can be assigned to object properties.

Change the value of a variable

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

   The Open Project dialog box opens.
3. Open the Variable Test project in the c:\vb08sbs\chap05\variable test folder.
4. If the project's form isn't visible, click Form1.vb in Solution Explorer, and then click the View Designer button.

   The Variable Test form opens in the Designer. Variable Test is a *skeleton program*—it contains a form with labels and buttons for displaying output, but little program code. (I create these skeleton programs now and then to save you time, although you can also create the project from scratch.) You'll add code in this exercise.

The Variable Test form looks like this:

The form contains two labels and two buttons. You'll use variables to display information in each of the labels.

Note The label objects look like boxes because I set their *BorderStyle* properties to Fixed3D.

5. Double-click the Show button.

   The *Button1_Click* event procedure appears in the Code Editor.

6. Type the following program statements to declare and use the *LastName* variable:

```
Dim LastName As String

LastName = "Luther"
Label1.Text = LastName

LastName = "Bodenstein von Karlstadt"
Label2.Text = LastName
```

The program statements are arranged in three groups. The first statement declares the *LastName* variable by using the *Dim* statement and the *String* type. After you type this line, Visual Studio places a green jagged line under the *LastName* variable, because it has been declared but not used in the program. There is nothing wrong here—Visual Studio is just reminding you that a new variable has been created and is waiting to be used.

Tip If the variable name still has a jagged underline when you finish writing your program, it could be a sign that you misspelled a variable name somewhere within your code.

The second and third lines assign the name "Luther" to the *LastName* variable and then display this name in the first label on the form. This example demonstrates one of the most common uses of variables in a program—transferring information to a property. As you have seen before, all string values assigned to variables are displayed in red type.

The fourth line assigns the name "Bodenstein von Karlstadt" to the *LastName* variable (in other words, it changes the contents of the variable). Notice that the second string is longer than the first and contains a few blank spaces. When you assign text strings to variables, or use them in other places, you need to enclose the text within quotation marks. (You don't need to do this with numbers.)

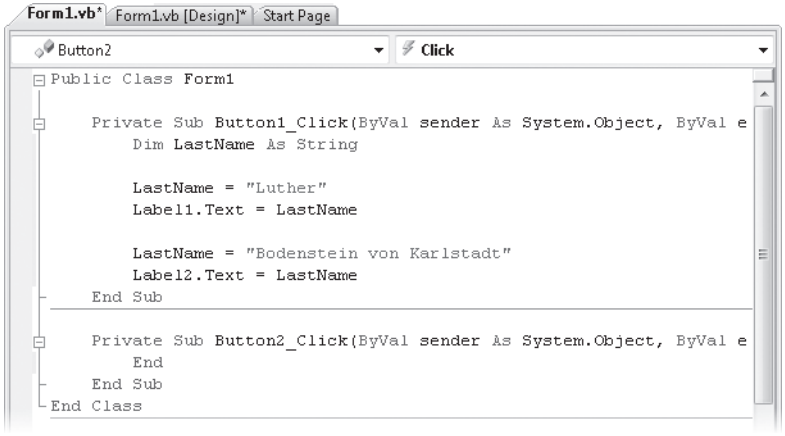
Finally, keep in mind another important characteristic of the variables being declared in this event procedure—they maintain their *scope*, or hold their value, only within the event procedure you're using them in. Later in this chapter, you'll learn how to declare variables so that they can be used in any of your form's event procedures.

7. Click the Form1.vb [Design] tab to display the form again.
8. Double-click the Quit button.

   The *Button2_Click* event procedure appears in the Code Editor.
9. Type the following program statement to stop the program:

```
End
```

   Your screen looks like this:



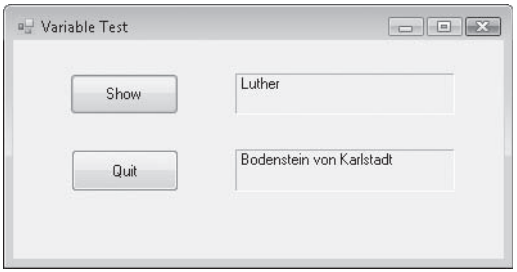
10. Click the Save All button on the Standard toolbar to save your changes.

11. Click the Start Debugging button on the Standard toolbar to run the program.

    The program runs in the IDE.

12. Click the Show button.

    The program declares the variable, assigns two values to it, and copies each value to the appropriate label on the form. The program produces the output shown in the following figure.



13. Click the Quit button to stop the program.

    The program stops, and the development environment returns.

Variable Naming Conventions

Naming variables can be a little tricky because you need to use names that are short but intuitive and easy to remember. To avoid confusion, use the following conventions when naming variables:

- Begin each variable name with a letter or underscore. This is a Visual Basic requirement. Variable names can contain only letters, underscores, and numbers.
- Although variable names can be virtually any length, try to keep them under 33 characters to make them easier to read. (Variable names are limited to 255 characters in Visual Basic 6, but that's no longer a constraint.)
- Make your variable names descriptive by combining one or more words when it makes sense to do so. For example, the variable name *SalesTaxRate* is much clearer than *Tax* or *Rate*.
- Use a combination of uppercase and lowercase characters and numbers. An accepted convention is to capitalize the first letter of each word in a variable; for example, *DateOfBirth*. However, some programmers prefer to use so-called camel casing (making the first letter of a variable name lowercase) to distinguish variable names from functions and module names, which usually begin with uppercase letters. Examples of camel casing include *dateOfBirth*, employeeName, and counter.

- Don't use Visual Basic keywords, objects, or properties as variable names. If you do, you'll get an error when you try to run your program.
- Optionally, you can begin each variable name with a two-character or three-character abbreviation corresponding to the type of data that's stored in the variable. For example, use strName to show that the Name variable contains string data. Although you don't need to worry too much about this detail now, you should make a note of this convention for later—you'll see it in parts of the Visual Studio documentation and in many of the advanced books about Visual Basic programming. (This convention and abbreviation scheme was originally created by Microsoft Distinguished Engineer Charles Simonyi and is sometimes called the Hungarian Naming Convention.)

Using a Variable to Store Input

One practical use for a variable is to temporarily hold information that was entered by the user. Although you can often use an object such as a list box or a text box to gather this information, at times you might want to deal directly with the user and save the input in a variable rather than in a property. One way to gather input is to use the *InputBox* function to display a dialog box on the screen and then use a variable to store the text the user types. You'll try this approach in the following example.

Get input by using the *InputBox* function

1. On the File menu, click Open Project.

   The Open Project dialog box opens.

2. Open the Input Box project in the c:\vb08sbs\chap05\input box folder.

   The Input Box project opens in the IDE. Input Box is a skeleton program.

3. If the project's form isn't visible, click Form1.vb in Solution Explorer, and then click the View Designer button.

   The form contains one label and two buttons. You'll use the *InputBox* function to get input from the user, and then you'll display the input in the label on the form.

4. Double-click the Input Box button.

   The *Button1_Click* event procedure appears in the Code Editor.

5. Type the following program statements to declare two variables and call the *InputBox* function:

```
Dim Prompt, FullName As String
Prompt = "Please enter your name."

FullName = InputBox(Prompt)
Label1.Text = FullName
```

This time, you're declaring two variables by using the *Dim* statement: *Prompt* and *FullName*. Both variables are declared using the *String* type. (You can declare as many variables as you want on the same line, as long as they are of the same type.) Note that in Visual Basic 6, this same syntax would have produced different results. *Dim* would create the *Prompt* variable using the *Variant* type (because no type was specified) and the *FullName* variable using the *String* type. But this logical inconsistency has been fixed in Visual Basic versions 2002 and later.

The second line in the event procedure assigns a text string to the *Prompt* variable. This message is used as a text argument for the *InputBox* function. (An *argument* is a value or an expression passed to a procedure or a function.) The next line calls the *InputBox* function and assigns the result of the call (the text string the user enters) to the *FullName* variable. *InputBox* is a special Visual Basic function that displays a dialog box on the screen and prompts the user for input. In addition to a prompt string, the *InputBox* function supports other arguments you might want to use occasionally. Consult the Visual Studio documentation for details.

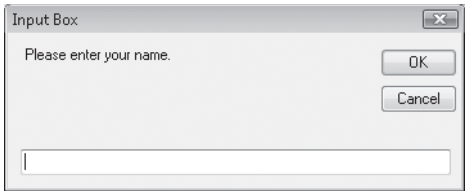
After *InputBox* has returned a text string to the program, the fourth statement in the procedure places the user's name in the *Text* property of the *Label1* object, which displays it on the form.

Note In older versions of BASIC, the InputBox function included a $ character at the end to help programmers remember that the function returned information in the string ($) data type. String variables were also identified with the $ symbol on occasion. These days we don't use character abbreviations for data types. String ($), Integer (%), and the other type abbreviations are now relics.

6. Save your changes.
7. Click the Start Debugging button on the Standard toolbar to run the program.

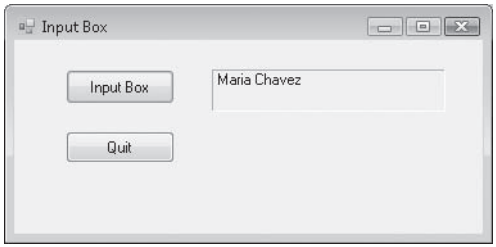
   The program runs in the IDE.
8. Click the Input Box button.

   Visual Basic executes the *Button1_Click* event procedure, and the Input Box dialog box opens on your screen, as shown here:



9. Type your full name, and then click OK.

   The *InputBox* function returns your name to the program and places it in the *FullName* variable. The program then uses the variable to display your name on the form, as shown here:



   Use the *InputBox* function in your programs anytime you want to prompt the user for information. You can use this function in combination with the other input controls to regulate the flow of data into and out of a program. In the next exercise, you'll learn how to use a similar function to display text in a dialog box.

10. Click the Quit button on the form to stop the program.

    The program stops, and the development environment reappears.

What Is a Function?

InputBox is a special Visual Basic keyword known as a *function*. A function is a statement that performs meaningful work (such as prompting the user for information or calculating an equation) and then returns a result to the program. The value returned by a function can be assigned to a variable, as it was in the Input Box program, or it can be assigned to a property or another statement or function. Visual Basic functions often use one or more arguments to define their activities. For example, the *InputBox* function you just executed used the *Prompt* variable to display dialog box instructions for the user. When a function uses more than one argument, commas separate the arguments, and the whole group of arguments is enclosed in parentheses. The following statement shows a function call that has two arguments:

```
FullName = InputBox(Prompt, Title)
```

Notice that I'm using italic in this syntax description to indicate that certain items are placeholders for information you specify. This is a style you'll find throughout the book and in the Visual Studio documentation.

Using a Variable for Output

You can display the contents of a variable by assigning the variable to a property (such as the *Text* property of a label object) or by passing the variable as an argument to a dialog box function. One useful dialog box function for displaying output is the *MsgBox* function. When you call the *MsgBox* function, it displays a dialog box, sometimes called a *message box*, with various options that you can specify. Like *InputBox*, it takes one or more arguments as input, and the results of the function call can be assigned to a variable. The syntax for the *MsgBox* function is

```
ButtonClicked = MsgBox(Prompt, Buttons, Title)
```

where *Prompt* is the text to be displayed in the message box; *Buttons* is a number that specifies the buttons, icons, and other options to display for the message box; and *Title* is the text displayed in the message box title bar. The variable *ButtonClicked* is assigned the result returned by the function, which indicates which button the user clicked in the dialog box.

If you're just displaying a message using the *MsgBox* function, the *ButtonClicked* variable, the assignment operator (=), the *Buttons* argument, and the *Title* argument are optional. You'll be using the *Title* argument, but you won't be using the others in the following exercise; for more information about them (including the different buttons you can include in *MsgBox* and a few more options), search for *MsgBox Function* in the Visual Studio documentation.

Note Visual Basic provides both the *MsgBox* function and the *MessageBox* class for displaying text in a message box. The *MessageBox* class is part of the *System.Windows.Forms* namespace, it takes arguments much like *MsgBox*, and it is displayed by using the *Show* method. I'll use both *MsgBox* and *MessageBox* in this book.

Now you'll add a *MsgBox* function to the Input Box program to display the name the user enters in the Input Box dialog box.

Display a message by using the *MsgBox* function

1. If the Code Editor isn't visible, double-click the Input Box button on the Input Box form.

   The *Button1_Click* event procedure appears in the Code Editor. (This is the code you entered in the last exercise.)

2. Select the following statement in the event procedure (the last line):

   ```
   Label1.Text = FullName
   ```

   This is the statement that displays the contents of the *FullName* variable in the label.

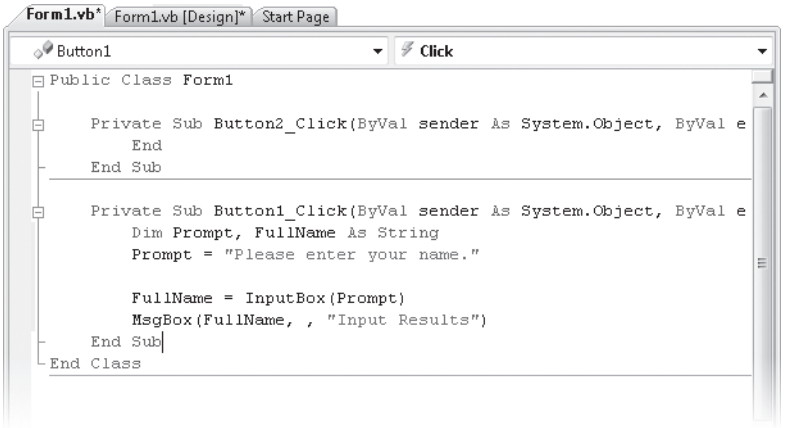
3. Press the Delete key to delete the line.

   The statement is removed from the Code Editor.

4. Type the following line into the event procedure as a replacement:

```
MsgBox(FullName, , "Input Results")
```

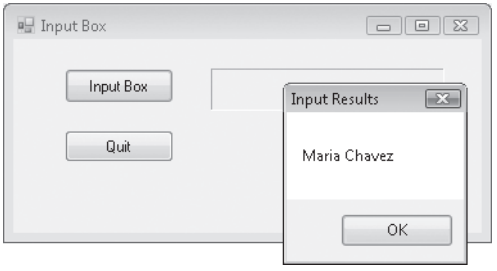
   This new statement will call the *MsgBox* function, display the contents of the *FullName* variable in the dialog box, and place the words *Input Results* in the title bar. (The optional *Buttons* argument and the *ButtonClicked* variable are irrelevant here and have been omitted.) Your event procedure looks like this:



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

6. Click the Input Box button, type your name in the input box, and then click OK.

   Visual Basic stores the input in the program in the *FullName* variable and then displays it in a message box. Your screen looks similar to this:



7. Click OK to close the message box. Then click Quit to close the program.

   The program closes, and the development environment returns.

Working with Specific Data Types

The *String* data type is useful for managing text in your programs, but what about numbers, dates, and other types of information? To allow for the efficient memory management of all types of data, Visual Basic provides several additional data types that you can use for your variables. Many of these are familiar data types from earlier versions of BASIC or Visual Basic, and some of the data types were introduced in Visual Studio 2005 to allow for the efficient processing of data in newer 64-bit computers.

The following table lists the fundamental (or elementary) data types in Visual Basic. Four new data types were added in Visual Basic 2005: *SByte*, *UShort*, *UInteger*, and *ULong*. *SByte* allows for "signed" byte values—that is, for both positive and negative numbers. *UShort*, *UInteger*, and *ULong* are "unsigned" data types—meaning that they cannot hold negative numbers. (However, as unsigned data types they offer twice the positive-number range of their signed counterparts, as shown in the table below.) If your program needs to perform a lot of calculations, you'll gain a performance advantage in your programs if you choose the right data type for your variables—a size that's neither too big nor too small. In the next exercise, you'll see how several of these data types work.

Note Variable storage size is measured in bits. The amount of space required to store one standard (ASCII) keyboard character in memory is 8 bits, which equals 1 byte.

Data type	Size	Range	Sample usage
Short	16-bit	-32,768 through 32,767	`Dim Birds As Short` `Birds = 12500`
UShort	16-bit	0 through 65,535	`Dim Days As UShort` `Days = 55000`
Integer	32-bit	-2,147,483,648 through 2,147,483,647	`Dim Insects As Integer` `Insects = 37500000`
UInteger	32-bit	0 through 4,294,967,295	`Dim Joys As UInteger` `Joys = 3000000000`
Long	64-bit	-9,223,372,036,854,775,808 to 9,223,372,036,854,775,807	`Dim WorldPop As Long` `WorldPop = 4800000004`

ULong	64-bit	0 through 18,446,744,073,709,551,615	`Dim Stars As ULong`
			`Stars = _`
			`18000000000000000000`

Single	32-bit floating point	-3.4028235E38 through 3.4028235E38	`Dim Price As Single` `Price = 899.99`
Double	64-bit floating point	-1.79769313486231E308 through 1.79769313486231E308	`Dim Pi As Double` `Pi = 3.1415926535`

Data type	Size	Range	Sample usage
Decimal	128-bit	0 through +/-79,228,162,514,264,337,593,543,950,335 (+/-7.9...E+28) with no decimal point; 0 through +/-7.9228162514264337593543950335 with 28 places to the right of the decimal. Append "D" if you want to force Visual Basic to initialize a *Decimal*.	`Dim Debt As Decimal` `Debt = 7600300.5D`
Byte	8-bit	0 through 255 (no negative numbers)	`Dim RetKey As Byte` `RetKey = 13`
SByte	8-bit	-128 through 127	`Dim NegVal As SByte` `NegVal = -20`
Char	16-bit	Any Unicode symbol in the range 0–65,535. Append "c" when initializing a *Char*.	`Dim UnicodeChar As Char` `UnicodeChar = " "c`
String	Usually 16-bits per character	0 to approximately 2 billion 16-bit Unicode characters	`Dim Dog As String` `Dog = "pointer"`
Boolean	16-bit	True or False. (During conversions, 0 is converted to False, other values to True.)	`Dim Flag as Boolean` `Flag = True`
Date	64-bit	January 1, 0001, through December 31, 9999	`Dim Birthday as Date` `Birthday = #3/1/1963#`

Object	32-bit	Any type can be stored in a variable of type *Object*.	`Dim MyApp As Object`
			`MyApp = CreateObject _`
			`("Word.Application")`

Use fundamental data types in code

1. On the File menu, click Open Project.

   The Open Project dialog box opens.

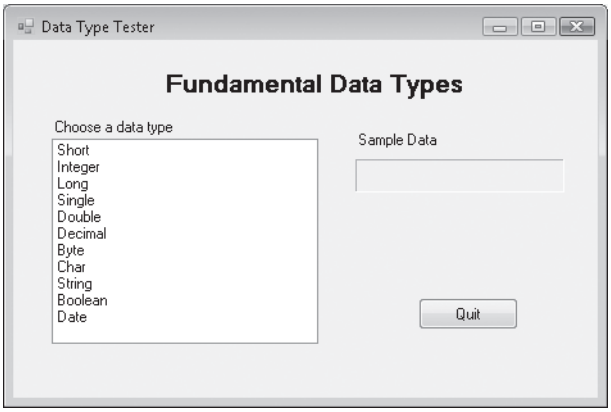
2. Open the Data Types project from the c:\vb08sbs\chap05\data types folder.

3. If the project's form isn't visible, click Form1.vb in Solution Explorer, and then click the View Designer button.

   Data Types is a complete Visual Basic program that I created to demonstrate how the fundamental data types work. You'll run the program to see what the data types look like, and then you'll look at how the variables are declared and used in the program code. You'll also learn where to place variable declarations so that they're available to all the event procedures in your program.

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

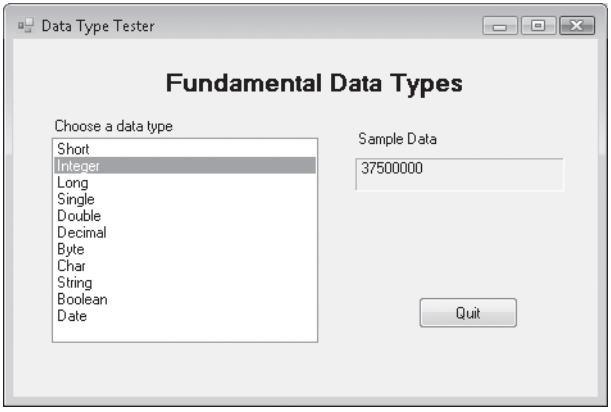
   The following application window opens:



   The Data Types program lets you experiment with 11 data types, including integer, single-precision floating point, and date. The program displays an example of each type when you click its name in the list box.

5. Click the *Integer* type in the list box.

   The number 37500000 appears in the Sample Data box. Note that with the *Short*, *Integer*, and *Long* data types, you can't insert or display commas. To display commas, you'll need to use the *Format* function.



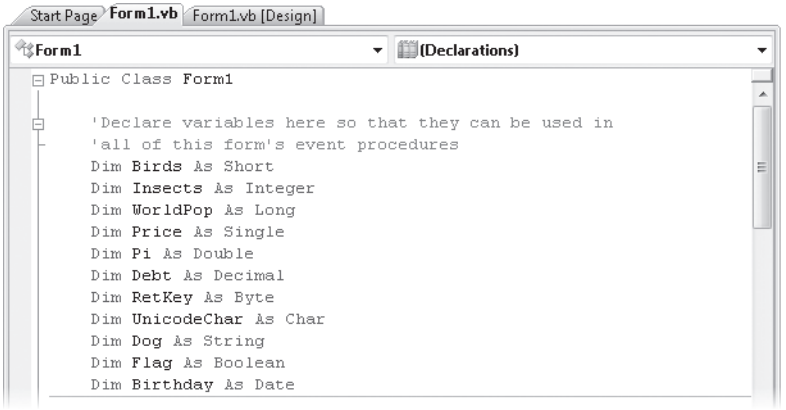
6. Click the *Date* type in the list box.

   The date 3/1/1963 appears in the Sample Data box.

7. Click each data type in the list box to see how Visual Basic displays it in the Sample Data box.
8. Click the Quit button to stop the program.

   Now you'll examine how the fundamental data types are declared at the top of the form and how they're used in the *ListBox1_SelectedIndexChanged* event procedure.
9. Double-click the form itself (not any objects on the form), and enlarge the Code Editor to see more of the program code.

   The Code Editor looks like this:

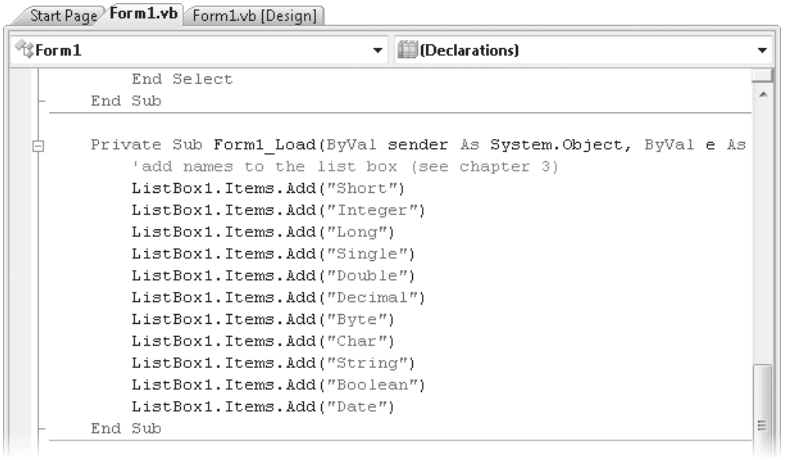


Scroll to the top of the Code Editor to see the dozen or so program statements I added to declare 11 variables in your program—one for each of the fundamental data types in Visual Basic. (I didn't create an example for the *SByte*, *UShort*, *UInteger*, and *ULong* types, because they closely resemble their signed or unsigned counterparts.) By placing each *Dim* statement here, at the top of the form's code initialization area, I'm ensuring that the variables will be valid, or will have *scope*, for all of the form's event procedures. That way, I can set the value of a variable in one event procedure and read it in another. Normally, variables are valid only in the event procedure in which they're declared. To make them valid across the form, you need to declare variables at the top of your form's code.

Note I've given each variable the same name as I did in the data types table earlier in the chapter so that you can see the examples I showed you in actual program code.

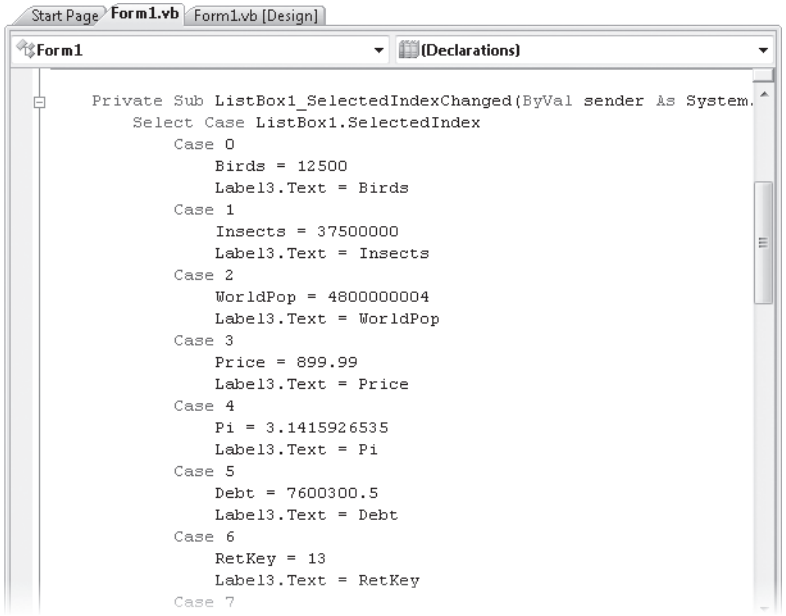
10. Scroll down in the Code Editor, and examine the *Form1_Load* event procedure.

You'll see the following statements, which add items to the list box object in the program. (You might remember this syntax from Chapter 3, "Working with Toolbox Controls"—I used some similar statements there.)



11. Scroll down and examine the *ListBox1_SelectedIndexChanged* event procedure.

The *ListBox1_SelectedIndexChanged* event procedure processes the selections you make in the list box and looks like this:



The heart of the event procedure is a *Select Case* decision structure. In the next chapter, we'll discuss how this group of program statements selects one choice from many. For now, notice how each section of the *Select Case* block assigns a sample value to one of the fundamental data type variables and then assigns the variable to the *Text* property of the *Label4* object on the form. I used code like this in Chapter 3 to process list box choices, and you can use these techniques to work with list boxes and data types in your own programs.

Note If you have more than one form in your project, you need to declare variables in a slightly different way (and place) to give them scope throughout your program (that is, in each form that your project contains). The type of variable that you'll declare is a public, or global, variable, and it's declared in a module, a special file that contains declarations and procedures not associated with a particular form. For information about creating public variables in modules, see Chapter 10, "Creating Modules and Procedures."

12. Scroll through the *ListBox1_SelectedIndexChanged* event procedure, and examine each of the variable assignments closely.

Try changing the data in a few of the variable assignment statements and running the program again to see what the data looks like. In particular, you might try assigning values to variables that are outside their accepted range, as shown in the data types table presented earlier. If you make such an error, Visual Basic adds a jagged line below the incorrect value in the Code Editor, and the program won't run until you change it. To learn more about your mistake, you can point to the jagged underlined value and read a short tooltip error message about the problem.

Tip By default, a green jagged line indicates a warning, a red jagged line indicates a syntax error, a blue jagged line indicates a compiler error, and a purple jagged line indicates some other error.

13. If you made any changes you want to save to disk, click the Save All button on the Standard toolbar.

User-Defined Data Types

Visual Basic also lets you create your own data types. This feature is most useful when you're dealing with a group of data items that naturally fit together but fall into different data categories. You create a *user-defined type* (UDT) by using the *Structure* statement, and you declare variables associated with the new type by using the *Dim* statement. Be aware that the *Structure* statement cannot be located in an event procedure—it must be located at the top of the form along with other variable declarations, or in a code module.

For example, the following declaration creates a user-defined data type named *Employee* that can store the name, date of birth, and hire date associated with a worker:

```
Structure Employee
    Dim Name As String
    Dim DateOfBirth As Date
    Dim HireDate As Date
End Structure
```

After you create a data type, you can use it in the program code for the form's or module's event procedures. The following statements use the new *Employee* type. The first statement creates a variable named *ProductManager*, of the *Employee* type, and the second statement assigns the name "Greg Baker" to the *Name* component of the variable:

```
Dim ProductManager As Employee
ProductManager.Name = "Greg Baker"
```

This looks a little similar to setting a property, doesn't it? Visual Basic uses the same notation for the relationship between objects and properties as it uses for the relationship between user-defined data types and component variables.

Constants: Variables That Don't Change

If a variable in your program contains a value that never changes (such as π, a fixed mathematical entity), you might consider storing the value as a constant instead of as a variable. A *constant* is a meaningful name that takes the place of a number or a text string that doesn't change. Constants are useful because they increase the readability of program code, they can reduce programming mistakes, and they make global changes easier to accomplish later. Constants operate a lot like variables, but you can't modify their values at run time. They are declared with the *Const* keyword, as shown in the following example:

```
Const Pi As Double = 3.14159265
```

This statement creates a constant named *Pi* that can be used in place of the value of π in the program code. To make a constant available to all the objects and event procedures in your form, place the statement at the top of your form along with other variable and structure declarations that will have scope in all of the form's event procedures. To make the constant available to all the forms and modules in a program (not just *Form1*), create the constant in a code module, with the *Public* keyword in front of it. For example:

```
Public Const Pi As Double = 3.14159265
```

The following exercise demonstrates how you can use a constant in an event procedure.

Use a constant in an event procedure

1. On the File menu, click Open Project.

   The Open Project dialog box opens.

2. Open the Constant Tester project in the c:\vb08sbs\chap05\constant tester folder.

3. If the project's form isn't visible, click Form1.vb in Solution Explorer, and then click the View Designer button.

   The Constant Tester form opens in the Designer. Constant Tester is a skeleton program. The user interface is finished, but you need to type in the program code.

4. Double-click the Show Constant button on the form.

   The *Button1_Click* event procedure appears in the Code Editor.

5. Type the following statements in the *Button1_Click* event procedure:

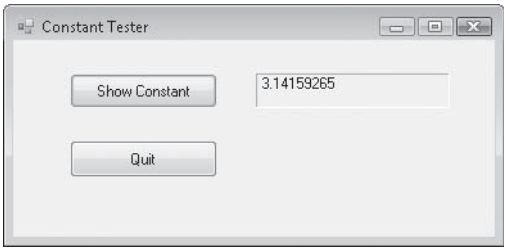
   ```
   Const Pi As Double = 3.14159265
   Label1.Text = Pi
   ```

Tip The location you choose for your declarations should be based on how you plan to use the constants or the variables. Programmers typically keep the scope for declarations as small as possible, while still making them available for code that needs to use them. For example, if a constant is needed only in a single event procedure, you should put the constant declaration within that event procedure. However, you could also place the declaration at the top of the form's code, which would give all the event procedures in your form access to it.

6. Click the Start Debugging button on the Standard toolbar to run the program.

7. Click the Show Constant button.

   The *Pi* constant appears in the label box, as shown here:



8. Click the Quit button to stop the program.

   Constants are useful in program code, especially in involved mathematical formulas, such as Area = πr^2. The next section describes how you can use operators and variables to write similar formulas.

Working with Visual Basic Operators

A *formula* is a statement that combines numbers, variables, operators, and keywords to create a new value. Visual Basic contains several language elements designed for use in formulas. In this section, you'll practice working with arithmetic (or mathematical) *operators*, the symbols used to tie together the parts of a formula. With a few exceptions, the arithmetic symbols you'll use are the ones you use in everyday life, and their operations are fairly intuitive. You'll see each operator demonstrated in the following exercises.

Visual Basic includes the following arithmetic operators:

Operator	Description
+	Addition
–	Subtraction
*	Multiplication
/	Division
\	Integer (whole number) division
Mod	Remainder division
^	Exponentiation (raising to a power)
&	String concatenation (combination)

Basic Math: The +, –, *, and / Operators

The operators for addition, subtraction, multiplication, and division are pretty straightforward and can be used in any formula where numbers or numeric variables are used. The following exercise demonstrates how you can use them in a program.

Work with basic operators

1. On the File menu, click Open Project.
2. Open the Basic Math project in the c:\vb08sbs\chap05\basic math folder.
3. If the project's form isn't visible, click Form1.vb in Solution Explorer, and then click the View Designer button.

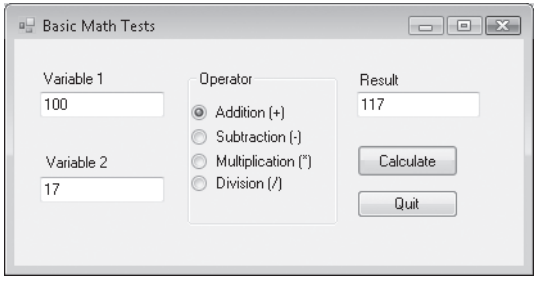
   The Basic Math form opens in the Designer. The Basic Math program demonstrates how the addition, subtraction, multiplication, and division operators work with numbers you type. It also demonstrates how you can use text box, radio button, and button objects to process user input in a program.
4. Click the Start Debugging button on the Standard toolbar.

   The Basic Math program runs in the IDE. The program displays two text boxes in which you enter numeric values, a group of operator radio buttons, a box that displays results, and two button objects (Calculate and Quit).
5. Type **100** in the Variable 1 text box, and then press Tab.

   The insertion point, or *focus*, moves to the second text box.
6. Type **17** in the Variable 2 text box.

   You can now apply any of the mathematical operators to the values in the text boxes.
7. Click the Addition radio button, and then click the Calculate button.

   The operator is applied to the two values, and the number 117 appears in the Result box, as shown in the following illustration.



8. Practice using the subtraction, multiplication, and division operators with the two numbers in the variable boxes. (Click Calculate to calculate each formula.)

   The results appear in the Result box. Feel free to experiment with different numbers in the variable text boxes. (Try a few numbers with decimal points if you like.) I used the *Double* data type to declare the variables, so you can use very large numbers.

   Now try the following test to see what happens:

9. Type **100** in the Variable 1 text box, type **0** in the Variable 2 text box, click the Division radio button, and then click Calculate.

   Dividing by zero is not allowed in mathematical calculations, because it produces an infinite result. But Visual Basic is able to handle this calculation and displays a value of Infinity in the Result text box. Being able to handle some divide-by-zero conditions is a feature that Visual Basic 2008 automatically provides.

10. When you've finished contemplating this and other tests, click the Quit button.

    The program stops, and the development environment returns.

Now take a look at the program code to see how the results were calculated. Basic Math uses a few of the standard input controls you experimented with in Chapter 3 and an event procedure that uses variables and operators to process the simple mathematical formulas. The program declares its variables at the top of the form so that they can be used in all of the Form1 event procedures.

Examine the Basic Math program code

1. Double-click the Calculate button on the form.

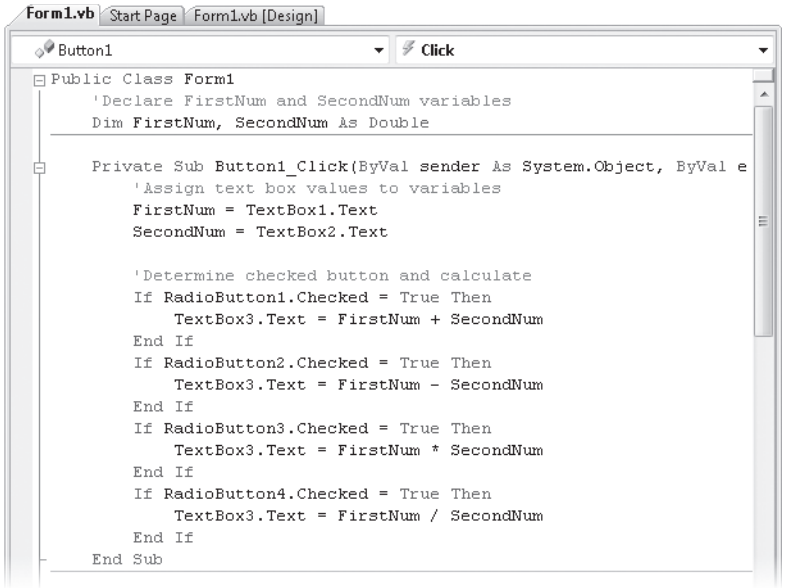
   The Code Editor displays the *Button1_Click* event procedure. At the top of the form's code, you'll see the following statement, which declares two variables of type *Double*:

   ```
   'Declare FirstNum and SecondNum variables
   Dim FirstNum, SecondNum As Double
   ```

   I used the *Double* type because I wanted a large, general purpose variable type that could handle many different numbers—integers, numbers with decimal points, very big numbers, small numbers, and so on. The variables are declared on the same line by using the shortcut notation. Both *FirstNum* and *SecondNum* are of type *Double*, and are used to hold the values input in the first and second text boxes, respectively.

2. Scroll down in the Code Editor to see the contents of the *Button1_Click* event procedure.

   Your screen looks similar to this:



The first two statements in the event procedure transfer data entered in the text box objects into the *FirstNum* and *SecondNum* variables.

```
'Assign text box values to variables
FirstNum = TextBox1.Text
SecondNum = TextBox2.Text
```

The *TextBox* control handles the transfer with the *Text* property—a property that accepts text entered by the user and makes it available for use in the program. I'll make frequent use of the *TextBox* control in this book. When it's set to multiline and resized, it can display many lines of text—even a whole file!

After the text box values are assigned to the variables, the event procedure determines which radio button has been selected, calculates the mathematical formula, and displays the result in a third text box. The first radio button test looks like this:

```
'Determine checked button and calculate
If RadioButton1.Checked = True Then
    TextBox3.Text = FirstNum + SecondNum
End If
```

Remember from Chapter 3 that only one radio button object in a group box object can be selected at any given time. You can tell whether a radio button has been selected by evaluating the *Checked* property. If it's True, the button has been selected. If the *Checked* property is False, the button has not been selected. After this simple test, you're ready to compute the result and display it in the third text box object. That's all there is to using basic arithmetic operators. (You'll learn more about the syntax of *If...Then* tests in Chapter 6, "Using Decision Structures.")

You're done using the Basic Math program.

Shortcut Operators

An interesting feature of Visual Basic is that you can use shortcut operators for mathematical and string operations that involve changing the value of an existing variable. For example, if you combine the + symbol with the = symbol, you can add to a variable without repeating the variable name twice in the formula. Thus, you can write the formula X = X + 6 by using the syntax X += 6. The following table shows examples of these shortcut operators.

Operation	Long-form syntax	Shortcut syntax
Addition (+)	X = X + 6	X += 6
Subtraction (-)	X = X – 6	X -= 6
Multiplication (*)	X = X * 6	X *= 6
Division (/)	X = X / 6	X /= 6
Integer division (\)	X = X \ 6	X \= 6
Exponentiation (^)	X = X ^ 6	X ^= 6
String concatenation (&)	X = X & "ABC"	X &= "ABC"

Using Advanced Operators: \, *Mod*, ^, and &

In addition to the four basic arithmetic operators, Visual Basic includes four advanced operators, which perform integer division (\), remainder division (*Mod*), exponentiation (^), and string concatenation (&). These operators are useful in special-purpose mathematical formulas and text processing applications. The following utility (a slight modification of the Basic Math program) shows how you can use each of these operators in a program.

Work with advanced operators

1. On the File menu, click Open Project.

   The Open Project dialog box opens.

2. Open the Advanced Math project in the c:\vb08sbs\chap05\advanced math folder.

3. If the project's form isn't visible, click Form1.vb in Solution Explorer, and then click the View Designer button.

   The Advanced Math form opens in the Designer. The Advanced Math program is identical to the Basic Math program, with the exception of the operators shown in the radio buttons and in the program.

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

   The program displays two text boxes in which you enter numeric values, a group of operator radio buttons, a text box that displays results, and two buttons.

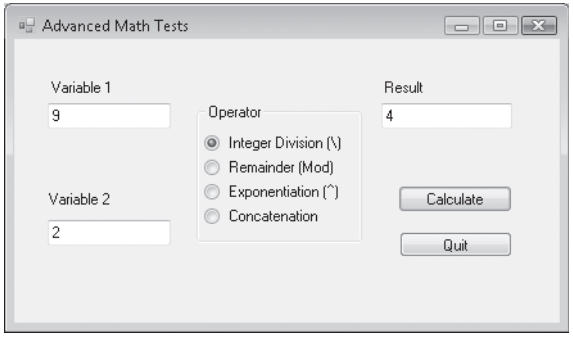
5. Type **9** in the Variable 1 text box, and then press Tab.

6. Type **2** in the Variable 2 text box.

   You can now apply any of the advanced operators to the values in the text boxes.

7. Click the Integer Division radio button, and then click the Calculate button.

   The operator is applied to the two values, and the number 4 appears in the Result box, as shown here:



   Integer division produces only the whole number result of the division operation. Although 9 divided by 2 equals 4.5, the integer division operation returns only the first part, an integer (the whole number 4). You might find this result useful if you're working with quantities that can't easily be divided into fractional components, such as the number of adults who can fit in a car.

8. Click the Remainder radio button, and then click the Calculate button.

   The number 1 appears in the Result box. Remainder division (modulus arithmetic) returns the remainder (the part left over) after two numbers are divided. Because 9 divided by 2 equals 4 with a remainder of 1 (2 * 4 + 1 = 9), the result produced by the *Mod* operator is 1. In addition to adding an early-seventies vibe to your code, the *Mod* operator can help you track "leftovers" in your calculations, such as the amount of money left over after a financial transaction.

9. Click the Exponentiation radio button, and then click the Calculate button.

   The number 81 appears in the Result box. The exponentiation operator (^) raises a number to a specified power. For example, 9 ^ 2 equals 9^2, or 81. In a Visual Basic formula, 9^2 is written 9 ^ 2.

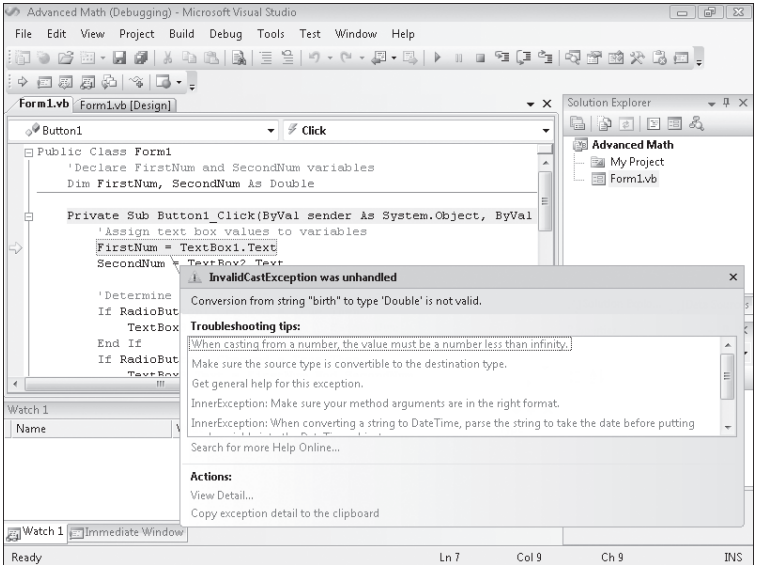
10. Click the Concatenation radio button, and then click the Calculate button.

The number 92 appears in the Result box. The string concatenation operator (*&*) combines two strings in a formula, but not through addition. The result is a combination of the "9" character and the "2" character. String concatenation can be performed on numeric variables—for example, if you're displaying the inning-by-inning score of a baseball game as they do in old-time score boxes—but concatenation is more commonly performed on string values or variables.

Because I declared the *FirstNum* and *SecondNum* variables as type *Double*, you can't combine words or letters by using the program code as written. As an example, try the following test, which causes an error and ends the program.

11. Type **birth** in the Variable 1 text box, type **day** in the Variable 2 text box, verify that Concatenation is selected, and then click Calculate.

Visual Basic is unable to process the text values you entered, so the program stops running, and an error message appears on the screen.



This type of error is called a *run-time error*—an error that surfaces not during the design and compilation of the program, but later, when the program is running and encounters a condition that it doesn't know how to process. If this seems odd, you might imagine that Visual Basic is simply offering you a modern rendition of the robot plea "Does not compute!" from the best science fiction films of the 1950s. The computer-speak message "Conversion from string "birth" to type 'Double' is not valid" means that the words you entered in the text boxes ("birth" and "day") could not be converted, or *cast*, by Visual Basic to variables of the type *Double*. *Double* types can only contain numbers. Period.

As we shall explore in more detail later, Visual Studio doesn't leave you hanging with such a problem, but provides a dialog box with different types of information to help you resolve the run-time error. For now, you have learned another important lesson about data types and when not to mix them.

12. Click the Stop Debugging button on the Standard toolbar to end the program.

Your program ends and returns you to the development environment.

Note In Chapter 8, "Debugging Visual Basic Programs," you'll learn about debugging mode, which allows you to track down the defects, or *bugs*, in your program code.

Now take a look at the program code to see how variables were declared and how the advanced operators were used.

13. Scroll to the code at the top of the Code Editor.

You see the following comment and program statement:

```
'Declare FirstNum and SecondNum variables
Dim FirstNum, SecondNum As Double
```

As you might recall from the previous exercise, *FirstNum* and *SecondNum* are the variables that hold numbers coming in from the *TextBox1* and *TextBox2* objects.

14. Change the data type from *Double* to *String* so that you can properly test how the string concatenation (*&*) operator works.

15. Scroll down in the Code Editor to see how the advanced operators are used in the program code.

You see the following code:

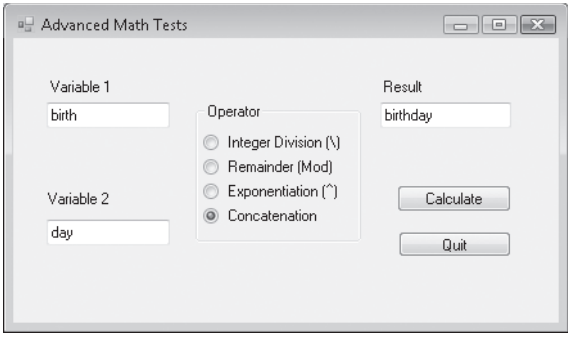
```
'Assign text box values to variables
FirstNum = TextBox1.Text
SecondNum = TextBox2.Text

'Determine checked button and calculate
If RadioButton1.Checked = True Then
    TextBox3.Text = FirstNum \ SecondNum
End If
If RadioButton2.Checked = True Then
    TextBox3.Text = FirstNum Mod SecondNum
End If
If RadioButton3.Checked = True Then
    TextBox3.Text = FirstNum ^ SecondNum
End If
If RadioButton4.Checked = True Then
    TextBox3.Text = FirstNum & SecondNum
End If
```

Like the Basic Math program, this program loads data from the text boxes and places it in the *FirstNum* and *SecondNum* variables. The program then checks to see which radio button the user checked and computes the requested formula. In this event procedure, the integer division (\), remainder (*Mod*), exponentiation (^), and string concatenation (*&*) operators are used. Now that you've changed the data type of the variables to *String*, run the program again to see how the *&* operator works on text.

16. Click the Start Debugging button.

17. Type **birth** in the Variable 1 text box, type **day** in the Variable 2 text box, click Concatenation, and then click Calculate.

    The program now concatenates the string values and doesn't produce a run-time error, as shown here:



18. Click the Quit button to close the program.

    You're finished working with the Advanced Math program.

Tip Run-time errors are difficult to avoid completely—even the most sophisticated application programs, such as Microsoft Word or Microsoft Excel, sometimes run into error conditions that they can't handle, producing run-time errors, or *crashes*. Designing your programs to handle many different data types and operating conditions helps you produce solid, or *robust*, applications. In Chapter 9, "Trapping Errors by Using Structured Error Handling," you'll learn about another helpful tool for preventing run-time error crashes—the structured error handler.

Working with Methods in the Microsoft .NET Framework

Now and then you'll want to do a little extra number crunching in your programs. You might need to round a number, calculate a complex mathematical expression, or introduce randomness into your programs. The math methods shown in the following table can help you work with numbers in your formulas. These methods are provided by the Microsoft .NET Framework, a class library that lets you tap into the power of the Windows operating system and accomplish many of the common programming tasks that you need to create your projects. The

.NET Framework is a major feature of Visual Studio that is shared by Visual Basic, Microsoft Visual C++, Microsoft Visual C#, and other tools in Visual Studio. It's an underlying interface that becomes part of the Windows operating system itself, and it is installed on each computer that runs Visual Studio programs.

The .NET Framework is organized into classes that you can use in your programming projects. The process is quite simple, and you'll experiment with how it works now by using a math method in the *System.Math* class of the .NET Framework.

What's New in Microsoft .NET Framework 3.5?

Visual Studio 2008 includes a new version of the .NET Framework—Microsoft .NET Framework 3.5. This is an update to the .NET Framework 3.0 software that provided support for the Windows Vista operating system, and the .NET Framework 2.0 software that shipped with Visual Studio 2005 and provided support for 64-bit processors. Version 3.5 adds new classes that provide additional functionality for distributed mobile applications, interprocess communication, time zone operations, ASP.NET, Visual Web Developer, and much more. The .NET Framework 3.5 also includes support for new advanced technologies, such as Language Integrated Query (LINQ) for querying different types of data, Windows Presentation Foundation (WPF) for creating complex graphics, Windows Communication Foundation (WCF) for creating applications that work with Web services, and Windows Workflow Foundation (WF) for creating workflow-type applications. Many of the improvements in the .NET Framework will come to you automatically as you use Visual Basic 2008, and some will become useful as you explore advanced programming techniques.

The following table offers a partial list of the math methods in the *System.Math* class. The argument *n* in the table represents the number, variable, or expression you want the method to evaluate. If you use any of these methods, be sure that you put the statement

```
Imports System.Math
```

at the very top of your form's code in the Code Editor.

Method	Purpose
Abs(n)	Returns the absolute value of *n*.
Atan(n)	Returns the arctangent, in radians, of *n*.
Cos(n)	Returns the cosine of the angle *n*. The angle *n* is expressed in radians.
Exp(n)	Returns the constant *e* raised to the power *n*.
Sign(n)	Returns -1 if *n* is less than 0, 0 if n is 0, and +1 if *n* is greater than 0.
Sin(n)	Returns the sine of the angle *n*. The angle *n* is expressed in radians.
Sqrt(n)	Returns the square root of *n*.
Tan(n)	Returns the tangent of the angle *n*. The angle *n* is expressed in radians.

Use the *System.Math* class to compute square roots

1. On the File menu, click New Project.

   The New Project dialog box opens.

2. Create a new Visual Basic Windows Forms Application project named **My Framework Math**.

   The new project is created, and a blank form opens in the Designer.

3. Click the *Button* control on the Windows Forms tab of the Toolbox, and create a button object at the top of your form.

4. Click the *TextBox* control in the Toolbox, and draw a text box below the button object.

5. Set the *Text* property of the button object to Square Root.

6. Double-click the button object to display the Code Editor.

7. At the very top of the Code Editor, above the *Public Class Form1* statement, type the following program statement:

   ```
   Imports System.Math
   ```

   The *System.Math* class is a collection of methods provided by the .NET Framework for arithmetic operations. The .NET Framework is organized in a hierarchical fashion and can be very deep. The *Imports* statement makes it easier to reference classes, properties, and methods in your project. For example, if you didn't include the previous *Imports* statement, to call the *Sqrt* method you would have to type *System.Math.Sqrt* instead of just *Sqrt*. The *Imports* statement must be the first statement in your program—it must come even before the variables that you declare for the form and the *Public Class Form1* statement that Visual Basic automatically provides.

8. Move down in the Code Editor, and add the following code to the *Button1_Click* event procedure between the *Private Sub* and *End Sub* statements:

   ```
   Dim Result As Double
   Result = Sqrt(625)
   TextBox1.Text = Result
   ```

   These three statements declare a variable of the double type named *Result*, use the *Sqrt* method to compute the square root of 625, and assign the *Result* variable to the *Text* property of the text box object so that the answer is displayed.

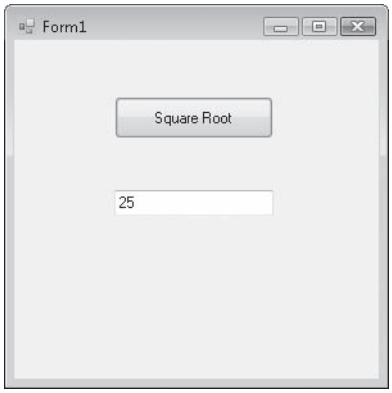
9. Click the Save All button on the Standard toolbar to save your changes. Specify the c:\vb08sbs\chap05 folder as the location.

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

    The Framework Math program runs in the IDE.

11. Click the Square Root button.

    Visual Basic calculates the square root of 625 and displays the result (25) in the text box. As you can see here, the *Sqrt* method works!



12. Click the Close button on the form to end the program.

To make it easier to reference classes, properties, and methods in the .NET Framework, include the *Imports* statement, and specify the appropriate namespace or class. You can use this technique to use any class in the .NET Framework, and you'll see many more examples of this technique as you work through *Microsoft Visual Basic 2008 Step by Step*.

One Step Further: Establishing Order of Precedence

In the previous few exercises, you experimented with several arithmetic operators and one string operator. Visual Basic lets you mix as many arithmetic operators as you like in a formula, as long as each numeric variable and expression is separated from another by one operator. For example, this is an acceptable Visual Basic formula:

```
Total = 10 + 15 * 2 / 4 ^ 2
```

The formula processes several values and assigns the result to a variable named *Total*. But how is such an expression evaluated by Visual Basic? In other words, what sequence does Visual Basic follow when solving the formula? You might not have noticed, but the order of evaluation matters a great deal in this example.

Visual Basic solves this dilemma by establishing a specific *order of precedence* for mathematical operations. This list of rules tells Visual Basic which operator to use first, second, and so on when evaluating an expression that contains more than one operator.

The following table lists the operators from first to last in the order in which they are evaluated. (Operators on the same level in this table are evaluated from left to right as they appear in an expression.)

Operator	Order of precedence
( )	Values within parentheses are always evaluated first.
^	Exponentiation (raising a number to a power) is second.
–	Negation (creating a negative number) is third.
*** / **	Multiplication and division are fourth.
\	Integer division is fifth.
Mod	Remainder division is sixth.
+ -	Addition and subtraction are last.

Given the order of precedence in this table, the expression

```
Total = 10 + 15 * 2 / 4 ^ 2
```

is evaluated by Visual Basic in the following steps. (Shading is used to show each step in the order of evaluation.)

```
Total = 10 + 15 * 2 / 4 ^ 2
Total = 10 + 15 * 2 / 16
Total = 10 + 30 / 16
Total = 10 + 1.875
Total = 11.875
```

Using Parentheses in a Formula

You can use one or more pairs of parentheses in a formula to clarify the order of precedence. For example, Visual Basic calculates the formula

```
Number = (8 - 5 * 3) ^ 2
```

by determining the value within the parentheses (-7) before doing the exponentiation—even though exponentiation is higher in order of precedence than subtraction and multiplication, according to the preceding table. You can further refine the calculation by placing nested parentheses in the formula. For example,

```
Number = ((8 - 5) * 3) ^ 2
```

directs Visual Basic to calculate the difference in the inner set of parentheses first, perform the operation in the outer parentheses next, and then determine the exponentiation. The result produced by the two formulas is different: the first formula evaluates to 49 and the second to 81. Parentheses can change the result of a mathematical operation, as well as make it easier to read.

Chapter 5 Quick Reference

To	Do this
Declare a variable	Type *Dim* followed by the variable name, the *As* keyword, and the variable data type in the program code. To make the variable valid in all of a form's event procedures, place this statement at the top of the code for the form, before any event procedures. For example: `Dim Country As String`
Change the value of a variable	Assign a new value with the assignment operator of (=). For example: `Country = "Japan"`
Get input by using a dialog box	Use the *InputBox* function, and assign the result to a variable. For example: `UserName = InputBox("What is your name?")`

Display output in a dialog box	Use the *MsgBox* function. (The string to be displayed in the dialog box can be stored in a variable.) For example:
	`Forecast = "Rain, mainly on the plain."`
	`MsgBox(Forecast, , "Spain Weather Report")`

Create a constant	Type the *Const* keyword followed by the constant name, the assignment operator (=), the constant data type, and the fixed value. For example: `Const JackBennysAge As Short = 39`
Create a formula	Link together numeric variables or values with one of the seven arithmetic operators, and then assign the result to a variable or a property. For example: `Result = 1 ^ 2 * 3 \ 4  'this equals 0`
Combine text strings	Use the string concatenation operator (*&*). For example: `Msg = "Hello" & "," & " world!"`
Make it easier to reference a class library from the .NET Framework	Place an *Imports* statement at the very top of the form's code that identifies the class library. For example: `Imports System.Math`
Make a call to a method from an included class library	Use the method name, and include any necessary arguments so that it can be used in a formula or a program statement. For example, to make a call to the *Sqrt* method in the *System.Math* class: `Hypotenuse = Sqrt(x ^ 2 + y ^ 2)`

Control the evaluation order in a formula	Use parentheses in the formula. For example:
	`Result = 1 + 2 ^ 3 \ 4       'this equals 3`
	`Result = (1 + 2) ^ ( 3 \ 4)  'this equals 1`