

# Chapter 4 *USING VARIABLES, DATA TYPES, AND CONSTANTS*

In programming, just as in life, certain things need to be done at once while others can be put off until later. When you postpone a task, you may enter it in your mental or paper “to-do” list. The individual entries on your list are often classified by their type or importance. When you delegate the task or finally get around to doing it yourself, you cross it off the list. This chapter shows you how your VBA procedures can memorize important pieces of information for use in later statements or calculations. You will learn how a procedure can keep a “to-do” entry in a variable, how variables are declared, and how they relate to data types and constants.

## SAVING RESULTS OF VBA STATEMENTS

---

In Chapter 3, while working in the Immediate window, you tried several Visual Basic instructions that returned some information. For example, when you entered `?Rows.Count`, you found out that there are 1,048,576 rows in a worksheet. However, when you write Visual Basic procedures outside of the Immediate window, you can't use the question mark. When you omit the question mark and enter `Rows.Count` in your procedure, Visual Basic won't stop suddenly to tell you the result of this instruction. If you want to know the result after executing a particular instruction, you must tell Visual Basic to memorize it. In programming, results returned by Visual Basic instructions can be written to variables.

## DATA TYPES

---

When you create Visual Basic procedures, you have a purpose in mind: You want to manipulate data. Because your procedures will handle different kinds of information, you should understand how Visual Basic stores data. The *data type* determines how the data is stored in the computer's memory. For example, data can be stored as a number, text, date, object, etc. If you forget to tell Visual Basic the type of your data, it assigns the Variant data type. The Variant type has the ability to figure out on its own what kind of data is being manipulated and then take on that type.

The Visual Basic data types are shown in Table 4.1. In addition to the built-in data types, you can define your own data types. You will see an example of a user-defined data type in Chapter 14, "Using Low-Level File Access" (see Hands-On 14.6 and "Understanding the Type Statement"). Because data types take up different amounts of space in the computer's memory, some of them are more expensive than others. Therefore, to conserve memory and make your procedure run faster, you should select the data type that uses the least amount of bytes and, at the same time, is capable of handling the data that your procedure has to manipulate.

## WHAT ARE VARIABLES?

---

A *variable* is simply a name that is used to refer to an item of data. Each time you want to remember a result of a VBA instruction, think of a name that will represent it. For example, if the number 1,048,576 has to remind you of the total number of rows in a worksheet (a very important piece of information when you want to bring external data into Excel 2010), you can make up a name such as `AllRows`, `NumOfRows`, or

**TABLE 4.1. VBA data types**

Data Type (Name)	Size (Bytes)	Description
Boolean	2	Stores a value of True (0) or False (–1)
Byte	1	A number in the range of 0 to 255
Integer	2	A number in the range of –32,768 to 32,767. The type declaration character for Integer is the percent sign (%).
Long	4	A number in the range of –2,147,483,648 to 2,147,483,647. The type declaration character for Long is the ampersand (&).
LongLong	8	Stored as a signed 64-bit (8-byte) number ranging in value from –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807. The type declaration character for LongLong is the caret (^). LongLong is a valid declared type only on 64-bit platforms.
LongPtr (Long integer on 32-bit systems; LongLong integer on 64-bit systems)	4 on 32-bit 8 on 64-bit	Numbers ranging in value from –2,147,483,648 to 2,147,483,647 on 32-bit systems; –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 on 64-bit systems. Using LongPtr enables writing code that can run in both 32-bit and 64-bit environments.
Single	4	Single-precision floating-point real number ranging in value from –3.402823E38 to –1.401298E–45 for negative values and from 1.401298E–45 to 3.402823E38 for positive values. The type declaration character for Single is the exclamation point (!).
Double	8	Double-precision floating-point real number in the range of –1.79769313486231E308 to –4.94065645841247E–324 for negative values and 4.94065645841247E–324 to 1.79769313486231E308 for positive values. The type declaration character for Double is the number sign (#).
Currency	8	(scaled integer) Monetary values used in fixed-point calculations: –922,337,203,685,477.5808 to 922,337,203,685,477.5807. The type declaration character for Currency is the at sign (@).

*Continued.*

Data Type (Name)	Size (Bytes)	Description
Decimal	14	<p>96-bit (12-byte) signed integer scaled by a variable power of 10. The power of 10 scaling factor specifies the number of digits to the right of the decimal point, and ranges from 0 to 28.</p> <p>With no decimal point (scale of 0), the largest value is <math>\pm 79,228,162,514,264,337,593,543,950,335</math>.</p> <p>With 28 decimal places, the largest value is <math>\pm 7.9228162514264337593543950335</math>.</p> <p>The smallest nonzero value is <math>\pm 0.000000000000000000000000000001</math>.</p> <p>You cannot declare a variable to be of type Decimal. You must use the Variant data type. Use the CDec function to convert a value to a decimal number:</p> <pre>Dim numDecimal As Variant numDecimal = CDec(0.02 * 15.75 * 0.0006)</pre>
Date	8	<p>Date from January 1, 100, to December 31, 9999, and times from 0:00:00 to 23:59:59. Date literals must be enclosed within number signs (#), for example: #January 1, 2011#</p>
String (variable-length)	10 bytes + string length	A variable-length string can contain up to approximately 2 billion characters. The type declaration character for String is the dollar sign (\$).
String (fixed-length)	Length of string	A fixed-length string can contain 1 to approximately 65,400 characters.
Object	4	Object variable used to refer to any Excel object. Use the Set statement to declare a variable as an Object.
Variant (with numbers)	16	Any numeric value up to the size of a Double.
Variant (with characters)	22 bytes + string length	Any valid nonnumeric data type in the same range as for a variable-length string.
User-Defined Data Type	One or more elements	<p>A data type you define using the Type statement. User-defined data types can contain one or more elements of a data type, an array, or a previously defined user-defined type. For example:</p> <pre>Type custInfo     custFullName as String     custTitle as String     custBusinessName as String     custFirstOrderDate as Date End Type</pre>

TotalRows, and so on. The names of variables can contain characters, numbers, and some punctuation marks, except for the following: , # \$ % & @ !

The name of a variable cannot begin with a number or contain a space. If you want the name of the variable to include more than one word, use the underscore (\_) as a separator. Although the name of a variable can contain as many as 254 characters, it's best to use short and simple variable names. Using short names will save you typing time when you need to refer to the variable in your Visual Basic procedure. Visual Basic doesn't care whether you use uppercase or lowercase letters in variable names, but most programmers use lowercase letters, and when the variable names are comprised of one or more words, they use title case, as in the names NumOfRows and First\_Name.

### Reserved Words Can't Be Used for Variable Names

---

You can use any label you want for a variable name, except for the reserved words that VBA uses. Visual Basic statements and certain other words that have a special meaning in VBA cannot be used as names of variables. For example, words such as Name, Len, Empty, Local, Currency, or Exit will generate an error message if used as a variable name.

---

### Meaningful Variable Names

---

Give variables names that can help you remember their roles. Some programmers use a prefix to identify the type of a variable. A variable name that begins with "str" (for example, strName) can be quickly recognized within the code of your procedure as the one holding the text string.

---

### How to Create Variables

You can create a variable by declaring it with a special command or by just using it in a statement. When you declare your variable, you make Visual Basic aware of the variable's name and data type. This is called *explicit variable declaration*. There are several advantages to explicit variable declaration:

- Explicit variable declaration speeds up the execution of your procedure. Since Visual Basic knows the data type, it reserves only as much memory as is absolutely necessary to store the data.
- Explicit variable declaration makes your code easier to read and understand because all the variables are listed at the very beginning of the procedure.
- Explicit variable declaration helps prevent errors caused by misspelled variable names. Visual Basic automatically corrects the variable name based on the spelling used in the variable declaration.

If you don't let Visual Basic know about the variable prior to using it, you are implicitly telling VBA that you want to create this variable. Variables declared implicitly are automatically assigned the Variant data type shown in Table 4.1. Although implicit variable declaration is convenient (it allows you to create variables on the fly and assign values without knowing in advance the data type of the values being assigned), it can cause several problems, as outlined below:

- If you misspell a variable name in your procedure, Visual Basic may display a runtime error or create a new variable. You are guaranteed to waste some time troubleshooting problems that could have been easily avoided had you declared your variable at the beginning of the procedure.
- Since Visual Basic does not know what type of data your variable will store, it assigns it a Variant data type. This causes your procedure to run slower because Visual Basic has to check the data type every time it deals with your variable. Because a Variant can store any type of data, Visual Basic has to reserve more memory to store your data.

## How to Declare Variables

You declare a variable with the `Dim` keyword. `Dim` stands for dimension. The `Dim` keyword is followed by the name of the variable and then the variable type.

Suppose you want the procedure to display the age of an employee. Before you can calculate the age, you must tell the procedure the employee's date of birth. To do this, you declare a variable called `DateOfBirth`, as follows:

```
Dim DateOfBirth As Date
```

Notice that the `Dim` keyword is followed by the name of the variable (`DateOfBirth`). This name can be anything you choose, as long as it is not one of the VBA keywords. Specify the data type the variable will hold by placing the `As` keyword after its name, followed by one of the data types from Table 4.1. The `Date` data type tells Visual Basic that the variable `DateOfBirth` will store a date. To store the employee's age, declare the age variable as follows:

```
Dim age As Integer
```

The `age` variable will store the number of years between today's date and the employee's date of birth. Since `age` is displayed as a whole number, this variable has been assigned the `Integer` data type.

You may also want your procedure to keep track of the employee's name, so you declare another variable to hold the employee's first and last name:

```
Dim FullName As String
```

Since the word “Name” is on the VBA list of reserved words, using it in your VBA procedure would guarantee an error. To hold the employee’s full name, call the variable `FullName` and declare it as the String data type, since the data it will hold is text.

Declaring variables is regarded as a good programming practice because it makes programs easier to read and helps prevent certain types of errors.

## Informal Variables

Variables that are not explicitly declared with `Dim` statements are said to be implicitly declared. These variables are automatically assigned a data type called Variant. They can hold numbers, strings, and other types of information. You can create a variable by simply assigning some value to a variable name anywhere in your VBA procedure. For example, you will implicitly declare a variable in the following way:

```
DaysLeft = 100.
```

Now that you know how to declare your variables, let’s take a look at a procedure that uses them:

```
Sub AgeCalc()  
    ' variable declaration  
    Dim FullName As String  
    Dim DateOfBirth As Date  
    Dim age As Integer  
  
    ' assign values to variables  
    FullName = "John Smith"  
    DateOfBirth = #01/03/1967#  
  
    ' calculate age  
    age = Year(Now()) - Year(DateOfBirth)  
  
    ' print results to the Immediate window  
    Debug.Print FullName & " is " & age & " years old."  
End Sub
```

The variables are declared at the beginning of the procedure in which they are going to be used. In this procedure, the variables are declared on separate lines. If you want, you can declare several variables on the same line, separating each variable name with a comma, as shown here:

```
Dim FullName As String, DateOfBirth As Date, age As Integer
```

Notice that the `Dim` keyword appears only once at the beginning of the variable declaration line.

When Visual Basic executes the variable declaration statements, it creates the variables with the specified names and reserves memory space to store their values. Then specific values are assigned to these variables.

To assign a value to a variable, begin with a variable name followed by an equals sign. The value entered to the right of the equals sign is the data you want to store in the variable. The data you enter here must be of the type determined by the variable declaration. Text data should be surrounded by quotation marks, and dates by the # characters.

Using the data supplied by the `DateOfBirth` variable, Visual Basic calculates the age of an employee and stores the result of the calculation in the `age` variable. Then the full name of the employee as well as the age is printed to the Immediate window using the instruction `Debug.Print`. When the Visual Basic procedure has executed, you must view the Immediate window to see the results.

Let's see what happens when you declare a variable with the incorrect data type. The purpose of the following procedure is to calculate the total number of rows in a worksheet and then display the results in a dialog box.

```
Sub HowManyRows()  
    Dim NumOfRows As Integer  
  
    NumOfRows = Rows.Count  
  
    MsgBox "The worksheet has " & NumOfRows & " rows."  
End Sub
```

A wrong data type can cause an error. In the procedure above, when Visual Basic attempts to write the result of the `Rows.Count` statement to the variable `NumOfRows`, the procedure fails and Excel displays the message “Run-time error 6 — Overflow.” This error results from selecting an invalid data type for that variable. The number of rows in a spreadsheet does not fit the Integer data range. To correct the problem, you should choose a data type that can accommodate a larger number:

```
Sub HowManyRows2()  
    Dim NumOfRows As Long  
  
    NumOfRows = Rows.Count  
    MsgBox "The worksheet has " & NumOfRows & " rows."  
End Sub
```

You can also correct the problem caused by the assignment of the wrong data type in the first example above by deleting the variable type (`As Integer`). When you rerun the procedure, Visual Basic will assign to your variable the Variant data type. Although Variants use up more memory than any other variable type and



also slow down the speed at which your procedures run (because Visual Basic has to do extra work to check the Variant's context), when it comes to short procedures, the cost of using Variants is barely noticeable.

## What Is the Variable Type?

You can quickly find out the type of a variable used in your procedure by right-clicking the variable name and selecting Quick Info from the shortcut menu.

## Concatenation

You can combine two or more strings to form a new string. The joining operation is called *concatenation*. You have seen examples of concatenated strings in the AgeCalc and HowManyRows2 procedures above. Concatenation is represented by an ampersand character (&). For instance, "His name is " & FirstName will produce the following string: His name is John. The name of the person is determined by the contents of the `FirstName` variable. Notice that there is an extra space between "is" and the ending quote: "His name is ". Concatenation of strings also can be represented by a plus sign (+). However, many programmers prefer to restrict the plus sign to operations on numbers to eliminate ambiguity.

## Specifying the Data Type of a Variable

If you don't specify the variable's data type in the `Dim` statement, you end up with an untyped variable. Untyped variables in VBA are always Variant data types. It's highly recommended that you create typed variables. When you declare a variable of a certain data type, your VBA procedure runs faster because Visual Basic does not have to stop to analyze the Variant variable to determine its type.

Visual Basic can work with many types of numeric variables. Integer variables can only hold whole numbers from -32,768 to 32,767. Other types of numeric variables are Long, Single, Double, and Currency. Long variables can hold whole numbers in the range -2,147,483,648 to 2,147,483,647. Unlike the Integer and Long variables, the Single and Double variables can hold decimals. String variables are used to refer to text. When you declare a variable of String data type, you can tell Visual Basic how long the string should be. For instance:

```
Dim extension As String * 3
```

declares a fixed-length String variable named `extension` that is three characters long. If you don't assign a specific length, the String variable will be dynamic. This

means that Visual Basic will make enough space in computer memory to handle whatever amount of text is assigned to it.

After you declare a variable, you can only store the type of information in it that you determined in the declaration statement. Assigning string values to numeric variables or numeric values to string variables results in the error message “Type mismatch” or causes Visual Basic to modify the value. For example, if your variable was declared to hold whole numbers and your data uses decimals, Visual Basic will disregard the decimals and use only the whole part of the number. When you run the MyNumber procedure shown below, Visual Basic modifies the data to fit the variable’s data type (Integer), and instead of 23.11 the variable ends up holding a value of 23.

```
Sub MyNumber()
    Dim myNum As Integer
    myNum = 23.11
    MsgBox myNum
End Sub
```

If you don’t declare a variable with a `Dim` statement, you can still designate a type for it by using a special character at the end of the variable name. To declare the `FirstName` variable as String, you can append the dollar sign to the variable name:

```
Dim FirstName$
```

This declaration is the same as `Dim FirstName As String`. The type declaration characters are shown in Table 4.2.

Notice that the type declaration characters can only be used with six data types. To use the type declaration character, append the character to the end of the variable name.

**TABLE 4.2. Type declaration characters**

Data Type	Character
Integer	%
Long	&
Single	!
Double	#
Currency	@
String	\$

In the AgeCalc2 procedure below we use two type declaration characters shown in Table 4.2.

```
Sub AgeCalc2()
    ' variable declaration
    Dim FullName$
    Dim DateOfBirth As Date
    Dim age%

    ' assign values to variables
    FullName$ = "John Smith"
    DateOfBirth = #1/3/1967#

    ' calculate age
    age% = Year(Now()) - Year(DateOfBirth)

    ' print results to the Immediate window
    Debug.Print FullName$ & " is " & age% & " years old."
End Sub
```

## Declaring Typed Variables

The variable type can be indicated by the `AS` keyword or a type symbol. If you don't add the type symbol or the `AS` command, the variable will be the default data type. VBA defaults to the Variant type.

## Assigning Values to Variables

Now that you know how to name and declare variables and have seen examples of using variables in complete procedures, let's gain experience using them. In Hands-On 4.1 we will begin by creating a variable and assigning it a specific value.



Please note files for the “Hands-On” project may be found on the companion CD-ROM.



### Hands-On 4.1. Writing a VBA Procedure with Variables

1. Open a new workbook and save it as `C:\Excel2010_ByExample\Practice_Excel04.xlsm`.
2. Activate the Visual Basic Editor window.
3. In the Project Explorer window, select the new project and change the name of the project in the Properties window to **Chapter4**.
4. Choose **Insert | Module** to add a new module to the **Chapter4 (Practice\_Excel04.xlsm)** VBA project.

5. In the Properties window, change the name of Module1 to **Variables**.
6. In the Code window, enter the CalcCost procedure shown below:

```
Sub CalcCost()  
    slsPrice = 35  
    slsTax = 0.085  
  
    Range("A1").Formula = "The cost of calculator"  
    Range("A4").Formula = "Price"  
    Range("B4").Formula = slsPrice  
    Range("A5").Formula = "Sales Tax"  
    Range("A6").Formula = "Cost"  
    Range("B5").Formula = slsPrice * slsTax  
    cost = slsPrice + (slsPrice * slsTax)  
  
    With Range("B6")  
        .Formula = cost  
        .NumberFormat = "0.00"  
    End With  
  
    strMsg = "The calculator total is $" & cost & "."  
    Range("A8").Formula = strMsg  
End Sub
```

The above procedure calculates the cost of purchasing a calculator using the following assumptions: The price of a calculator is 35 dollars and the sales tax equals 8.5%.

The procedure uses four variables: `slsPrice`, `slsTax`, `cost`, and `strMsg`. Because none of these variables have been explicitly declared, they all have the same data type — Variant. The variables `slsPrice` and `slsTax` were created by assigning some values to variable names at the beginning of the procedure. The `cost` variable was assigned a value that is a result of a calculation: `slsPrice + (slsPrice * slsTax)`. The cost calculation uses the values supplied by the `slsPrice` and `slsTax` variables. The `strMsg` variable puts together a text message to the user. This message is then entered as a complete sentence in a worksheet cell. When you assign values to variables, place an equals sign after the name of the variable. After the equals sign, you should enter the value of the variable. This can be a number, a formula, or text surrounded by quotation marks. While the values assigned to the variables `slsPrice`, `slsTax`, and `cost` are easily understood, the value stored in the `strMsg` variable is a little more involved. Let's examine the contents of the `strMsg` variable.

```
strMsg = "The calculator total is $ " & cost & "."
```

- The string "The calculator total is " is surrounded by quotation marks. Notice that there is an extra space before the ending quote.

- The dollar sign inside the quotes is used to denote the Currency data type. Because the dollar symbol is a character, it is surrounded by the quotes.
- The & character allows another string or the contents of a variable to be appended to the string. The & character must be used every time you want to append a new piece of information to the previous string.
- The `cost` variable is a placeholder. The actual cost of the calculator will be displayed here when the procedure runs.
- The & character attaches yet another string.
- The period is surrounded by quotes. When you require a period at the end of a sentence, you must attach it separately when it follows the name of the variable.

## Variable Initialization

When Visual Basic creates a new variable, it initializes the variable. Variables assume their default value. Numerical variables are set to zero (0), Boolean variables are initialized to False, String variables are set to the empty string (""), and Date variables are set to December 30, 1899.

Now let's execute the `CalcCost` procedure.

7. Position the cursor anywhere within the `CalcCost` procedure and choose **Run | Run Sub/UserForm**.

When you run this procedure, Visual Basic may display the following message: "Compile error: Variable not defined." If this happens, click **OK** to close the message box. Visual Basic will select the `slsPrice` variable and highlight the name of the `CalcCost` procedure. The title bar displays "Microsoft Visual Basic – Practice\_Excel04.xlsm [break]." The Visual Basic break mode allows you to correct the problem before you continue. Later in this book, you will learn how to fix problems in break mode. For now, exit this mode by choosing **Run | Reset**. Now go to the top of the Code window and delete the statement `Option Explicit` that appears on the first line. The `Option Explicit` statement means that all variables used within this module must be formally declared. You will learn about this statement in the next section. When the `Option Explicit` statement is removed from the Code window, choose **Run | Run Sub/UserForm** to rerun the procedure. This time, Visual Basic goes to work with no objections.

8. After the procedure has finished executing, press **Alt-F11** to switch to Microsoft Excel.  
The result of the procedure should match Figure 4.1.

	A	B	C	D
1	The cost of calculator			
2				
3				
4	Price	35		
5	Sales Tax	2.975		
6	Cost	37.98		
7				
8	The calculator total is \$37.975.			
9				

**FIGURE 4.1.** The VBA procedure can enter data and calculate results in a worksheet.

Cell A8 displays the contents of the `strMsg` variable. Notice that the cost entered in cell B6 has two decimal places, while the cost in `strMsg` displays three decimals. To display the cost of a calculator with two decimal places in cell A8, you must apply the required format not to the cell but to the `cost` variable itself.

VBA has special functions that allow you to change the format of data. To change the format of the `cost` variable, you will now use the `Format` function. This function has the following syntax:

```
Format(expression, format)
```

where `expression` is a value or variable that you want to format and `format` is the type of format you want to apply.

9. Change the calculation of the `cost` variable in the `CalcCost` procedure:

```
cost = Format(slsPrice + (slsPrice * slsTax), "0.00")
```

10. Replace the `With...End With` block of instructions with the following:

```
Range("B6").Formula = cost
```

11. Replace the statement `Range("B5").Formula = slsPrice * slsTax` with the following instruction:

```
Range("B5").Formula = Format((slsPrice * slsTax), "0.00")
```

12. Rerun the modified procedure.

Notice that now the text displayed in cell A8 shows the cost of the calculator formatted with two decimal places.

After trying out the CalcCost procedure, you may wonder why you should bother declaring variables if Visual Basic can handle undeclared variables so well. The CalcCost procedure is very short, so you don't need to worry about how many bytes of memory will be consumed each time Visual Basic uses the Variant variable. In short procedures, however, it is not the memory that matters but the mistakes you are bound to make when typing variable names. What will happen if the second time you use the `cost` variable you omit the "o" and refer to it as `cst`?

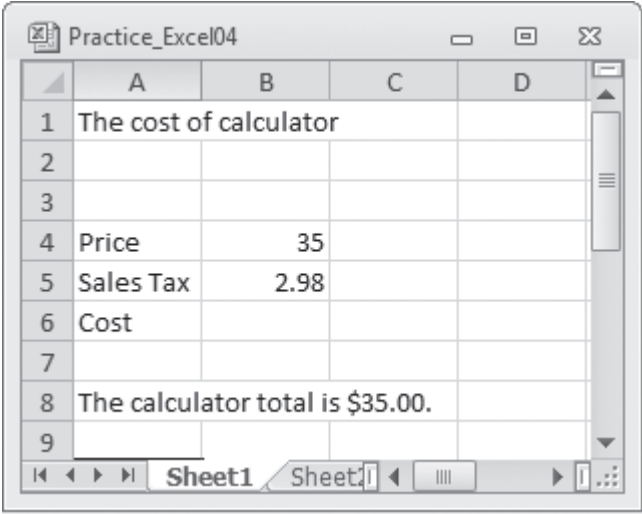
```
Range("B6").Formula = cst
```

What will you end up with if instead of `slsTax` you use the word `Tax` in the formula?

```
Cost = Format(slsPrice + (slsPrice * Tax), "0.00")
```

The result of the CalcCost procedure after introducing these two mistakes is shown in Figure 4.2.

Notice that in Figure 4.2 cell B6 does not show a value because Visual Basic does not find the assignment statement for the `cst` variable. Because Visual Basic does not know the sales tax, it displays the price of the calculator (see cell A8) as the total cost. Visual Basic does not guess. It simply does what you tell it to do. This brings us to the next section, which explains how to make sure this kind of error doesn't occur.



	A	B	C	D
1	The cost of calculator			
2				
3				
4	Price	35		
5	Sales Tax	2.98		
6	Cost			
7				
8	The calculator total is \$35.00.			
9				

**FIGURE 4.2.** Mistakes in the names of variables can produce incorrect results.

Note:

*If you have made changes in the variable names as described above, be sure to replace the names of the variables `cst` and `tax` with `cost` and `slsTax` in the appropriate lines of the VBA code before you continue.*

## CONVERTING BETWEEN DATA TYPES

While VBA handles a lot of data type conversion automatically in the background, it also provides a number of data conversion functions (see Table 4.3) that allow you to convert one data type to another. These functions should be used in situations where you want to show the result of an operation as a particular data type rather than the default data type. For example, instead of showing the result of your calculation as an Integer, single-precision or double-precision number, you may want to use the `CCur` function to force currency arithmetic, as in the following example procedure:

```
Sub ShowMoney()
    'declare variables of two different types
    Dim myAmount As Single
    Dim myMoneyAmount As Currency

    myAmount = 345.34

    myMoneyAmount = CCur(myAmount)
    Debug.Print "Amount = $" & myMoneyAmount
End Sub
```

When using the `CCur` function, currency options are recognized depending on the locale setting of your computer. The same holds true for the `CDate` function. By using this function you can ensure that the date is formatted according to the locale setting of your system. Use the `IsDate` function to determine whether a return value can be converted to date or time.

```
Sub ConvertToDate()
    'assume you have entered Jan 1 2011 in cell A1
    Dim myEntry As String
    Dim myRangeValue As Date

    myEntry = Sheet2.Range("A1").Value
    If IsDate(myEntry) Then
        myRangeValue = CDate(myEntry)
    End If
    Debug.Print myRangeValue
End Sub
```



**TABLE 4.3. VBA data type conversion functions**

Conversion Function	Return Type	Description														
CBool	Boolean	Any valid string or numeric expression														
CByte	Byte	0 to 255														
CCur	Currency	–922,337,203,685,477.5808 to 922,337,203,685,477.5807														
CDate	Date	Any valid date expression														
Cdbl	Double	–1.79769313486231E308 to –4.94065645841247E-324 for negative values; 4.94065645841247E-324 to 1.79769313486232E308 for positive values.														
CDec	Decimal	+/–79,228,162,514,264,337,593,543,950,335 for zero-scaled numbers, that is, numbers with no decimal places. For numbers with 28 decimal places, the range is +/-7.9228162514264337593543950335. The smallest possible nonzero number is 0.00000000000000000000000000000001.														
CInt	Integer	–32,768 to 32,767; fractions are rounded.														
CLng	Long	–2,147,483,648 to 2,147,483,647; fractions are rounded.														
CLngLng	LongLong	–9,223,372,036,854,775,808 to 9,223,372,036,854,775,807; fractions are rounded. (Valid on 64-bit platforms only.)														
CLngPtr	LongPtr	–2,147,483,648 to 2,147,483,647 on 32-bit systems; –9,223,372,036,854,775,808 to 9,223,372,036,854,775,807 on 64-bit systems. Fractions are rounded for 32-bit and 64-bit systems.														
CSng	Single	–3.402823E38 to –1.401298E-45 for negative values; 1.401298E-45 to 3.402823E38 for positive values.														
CStr	String	Returns for CStr depend on the expression argument. <table><tr><td>If Expression Is</td><td>CStr returns</td></tr><tr><td>Boolean</td><td>A string containing True or False</td></tr><tr><td>Date</td><td>A string containing a date in the short date format of your system</td></tr><tr><td>Null</td><td>A runtime error</td></tr><tr><td>Empty</td><td>A zero-length string (“”)</td></tr><tr><td>Error</td><td>A string containing the word “Error” followed by the error number</td></tr><tr><td>Other numeric</td><td>A string containing the number</td></tr></table>	If Expression Is	CStr returns	Boolean	A string containing True or False	Date	A string containing a date in the short date format of your system	Null	A runtime error	Empty	A zero-length string (“”)	Error	A string containing the word “Error” followed by the error number	Other numeric	A string containing the number
If Expression Is	CStr returns															
Boolean	A string containing True or False															
Date	A string containing a date in the short date format of your system															
Null	A runtime error															
Empty	A zero-length string (“”)															
Error	A string containing the word “Error” followed by the error number															
Other numeric	A string containing the number															
CVar	Variant	Same range as Double for numerics. Same range as String for nonnumerics.														

In cases where you need to round the value to the nearest even number, you will find the `CInt` and `CIntg` functions quite handy, as demonstrated in the following procedure:

```
Sub ShowInteger()  
    'declare variables of two different types  
    Dim myAmount As Single  
    Dim myIntAmount As Integer  
  
    myAmount = 345.64  
  
    myIntAmount = CInt(myAmount)  
    Debug.Print "Original Amount = " & myAmount  
    Debug.Print "New Amount = " & myIntAmount  
End Sub
```

As you can see in the code of the above procedures, the syntax for the VBA conversion functions is as follows:

```
conversionFunctionName(variableName)
```

where `variableName` is the name of a variable, a constant, or an expression (like `x + y`) that evaluates to a particular data type.

## Forcing Declaration of Variables

Visual Basic has the `Option Explicit` statement that automatically reminds you to formally declare all your variables. This statement has to be entered at the top of each of your modules. The `Option Explicit` statement will cause Visual Basic to generate an error message when you try to run a procedure that contains undeclared variables as demonstrated in Hands-On 4.2.



### Hands-On 4.2. Writing a VBA Procedure with Explicitly Declared Variables

This Hands-On requires prior completion of Hands-On 4.1.

1. Return to the Code window where you entered the `CalcCost` procedure.
2. At the top of the module window (in the first line), type **Option Explicit** and press **Enter**. Excel will display the statement in blue.
3. Run the `CalcCost` procedure. Visual Basic displays the error message "Compile error: Variable not defined."
4. Click **OK** to exit the message box.

Visual Basic highlights the name of the variable `slsPrice`. Now you have to formally declare this variable. When you declare the `slsPrice` variable and rerun your procedure, Visual Basic will generate the same error as soon as it encounters another variable name that was not declared.

5. Choose **Run | Reset** to reset the VBA project.
6. Enter the following declarations at the beginning of the `CalcCost` procedure:

```
' declaration of variables
Dim slsPrice As Currency
Dim slsTax As Single
Dim cost As Currency
Dim strMsg As String
```

The revised `CalcCost` procedure is shown below:

```
Sub CalcCost()
    ' declaration of variables
    Dim slsPrice As Currency
    Dim slsTax As Single
    Dim cost As Currency
    Dim strMsg As String

    slsPrice = 35
    slsTax = 0.085
    Range("A1").Formula = "The cost of calculator"
    Range("A4").Formula = "Price"
    Range("B4").Formula = slsPrice
    Range("A5").Formula = "Sales Tax"
    Range("A6").Formula = "Cost"
    Range("B5").Formula = Format((slsPrice * slsTax), "0.00")
    cost = Format(slsPrice + (slsPrice * slsTax), "0.00")

    Range("B6").Formula = cost
    strMsg = "The calculator total is $" & cost & "."
    Range("A8").Formula = strMsg
End Sub
```

7. Rerun the procedure to ensure that Excel no longer displays the error.

## Option Explicit in Every Module

To automatically include `Option Explicit` in every new module you create, follow these steps:

1. Choose **Tools | Options**.
2. Make sure that the **Require Variable Declaration** check box is selected in the **Options** dialog box (**Editor** tab).
3. Choose **OK** to close the **Options** dialog box.

From now on, every new module will be added with the `Option Explicit` statement in line 1. If you want to require variables to be explicitly declared in a previously created module,

you must enter the `Option Explicit` statement manually by editing the module yourself.

`Option Explicit` forces formal (explicit) declaration of all variables in a particular module. One big advantage of using `Option Explicit` is that any mistyping of the variable name will be detected at compile time (when Visual Basic attempts to translate the source code to executable code). If included, the `Option Explicit` statement must appear in a module before any procedures.

## Understanding the Scope of Variables

Variables can have different ranges of influence in a VBA procedure. The term *scope* defines the availability of a particular variable to the same procedure, other procedures, and other VBA projects.

Variables can have the following three levels of scope in Visual Basic for Applications:

- Procedure-level scope
- Module-level scope
- Project-level scope

### Procedure-Level (Local) Variables

From this chapter, you already know how to declare a variable by using the `Dim` keyword. The position of the `Dim` keyword in the module sheet determines the scope of a variable. Variables declared with the `Dim` keyword placed within a VBA procedure have a *procedure-level scope*.

Procedure-level variables are frequently referred to as *local variables*. Local variables can only be used in the procedure in which they were declared. Undeclared variables always have a procedure-level scope. A variable's name must be unique within its scope. This means that you cannot declare two variables with the same name in the same procedure. However, you can use the same variable name in different procedures. In other words, the `CalcCost` procedure can have the `slsTax` variable, and the `ExpenseRep` procedure in the same module can have its own variable called `slsTax`. Both variables are independent of each other.

### Module-Level Variables

Local variables help save computer memory. As soon as the procedure ends, the variable dies and Visual Basic returns the memory space used by the variable to the computer. In programming, however, you often want the variable to be available to other VBA procedures after the procedure in which the variable was declared has finished running. This situation requires that you change the scope of a variable. Instead of a procedure-level variable, you want to declare a module-level variable. To declare a module-level variable, you must place the `Dim` keyword at the top of the module sheet before any procedures (just below the `Option Explicit` keyword).

For instance, to make the `slsTax` variable available to any other procedure in the Variables module, declare the `slsTax` variable in the following way:

```
Option Explicit
Dim slsTax As Single

Sub CalcCost()
...Instructions of the procedure...
End Sub
```

In the example above, the `Dim` keyword is located at the top of the module, below the `Option Explicit` statement. Before you can see how this works, you need another procedure that uses the `slsTax` variable. In Hands-On 4.3 we will write a new VBA procedure named `ExpenseRep`.



#### Hands-On 4.3. Writing another VBA Procedure with a Module-Level Variable

1. In the Code window, cut the declaration line **`Dim slsTax As Single`** in the Variables module from the `CalcCost` procedure and paste it at the top of the module sheet below the `Option Explicit` statement.
2. In the same module where the `CalcCost` procedure is located, enter the code of the `ExpenseRep` procedure as shown below:

```
Sub ExpenseRep()
    Dim slsPrice As Currency
    Dim cost As Currency

    slsPrice = 55.99

    cost = slsPrice + (slsPrice * slsTax)
    MsgBox slsTax
    MsgBox cost
End Sub
```

The `ExpenseRep` procedure declares two `Currency` type variables: `slsPrice` and `cost`. The `slsPrice` variable is then assigned a value of 55.99. The `slsPrice` variable is independent of the `slsPrice` variable that is declared within the `CalcCost` procedure.

The `ExpenseRep` procedure calculates the cost of a purchase. The cost includes the sales tax stored in the `slsTax` variable. Because the sales tax is the same as the one used in the `CalcCost` procedure, the `slsTax` variable has been declared at the module level.

3. Run the **`ExpenseRep`** procedure.  
Because you have not yet run the `CalcCost` procedure, Visual Basic does not know the value of the `slsTax` variable, so it displays zero in the first message box.

4. Run the **CalcCost** procedure.

After Visual Basic executes the CalcCost procedure that you revised in Hands-On 4.2, the contents of the `sIsTax` variable equals 0.085. If `sIsTax` were a local variable, the contents of this variable would be empty upon the termination of the CalcCost procedure.

When you run the CalcCost procedure, Visual Basic erases the contents of all the variables except for the `sIsTax` variable, which was declared at a module level.

5. Run the **ExpenseRep** procedure again.

As soon as you attempt to calculate the cost by running the ExpenseRep procedure, Visual Basic retrieves the value of the `sIsTax` variable and uses it in the calculation.

## Private Variables

When you declare variables at a module level, you can use the `Private` keyword instead of the `Dim` keyword. For instance:

```
Private sIsTax As Single
```

Private variables are available only to the procedures that are part of the module where they were declared. Private variables are always declared at the top of the module after the `Option Explicit` statement.

## Keeping the Project-Level Variable Private

To prevent a project-level variable's contents from being referenced outside its project, you can use the `Option Private Module` statement at the top of the module sheet, just below the `Option Explicit` statement and before the declaration line. For example:

```
Option Explicit
Option Private Module
Public sIsTax As Single

Sub CalcCost()
...Instructions of the procedure...
End Sub
```

## Project-Level Variables

Module-level variables that are declared with the `Public` keyword (instead of `Dim`) have project-level scope. This means that they can be used in any Visual Basic for

Applications module. When you want to work with a variable in all the procedures in all the open VBA projects, you must declare it with the `Public` keyword. For instance:

```
Option Explicit
Public sIsTax As Single
Sub CalcCost()
...Instructions of the procedure...
End Sub
```

Notice that the `sIsTax` variable declared at the top of the module with the `Public` keyword will now be available to any other procedure or VBA project.

### Lifetime of Variables

In addition to scope, variables have a lifetime. The *lifetime* of a variable determines how long a variable retains its value. Module-level and project-level variables preserve their values as long as the project is open. Visual Basic, however, can reinitialize these variables if required by the program's logic. Local variables declared with the `Dim` statement lose their values when a procedure has finished. Local variables have a lifetime as long as a procedure is running, and they are reinitialized every time the program is run. Visual Basic allows you to extend the lifetime of a local variable by changing the way it is declared.

### Understanding and Using Static Variables

A variable declared with the `Static` keyword is a special type of local variable. Static variables are declared at the procedure level. Unlike local variables declared with the `Dim` keyword, static variables do not lose their contents when the program is not in their procedure. For example, when a VBA procedure with a static variable calls another procedure, after Visual Basic executes the statements of the called procedure and returns to the calling procedure, the static variable still retains the original value. The `CostOfPurchase` procedure shown in Hands-On 4.4 demonstrates the use of the static variable named `allPurchase`. Notice how this variable keeps track of the running total.



#### Hands-On 4.4. Writing a VBA Procedure with a Static Variable

1. In the Code window of the Variables module, write the following procedure:

```
Sub CostOfPurchase()
' declare variables
Static allPurchase
Dim newPurchase As String
```

```

Dim purchCost As Single

newPurchase = InputBox("Enter the cost of a purchase:")
purchCost = CSng(newPurchase)
allPurchase = allPurchase + purchCost

' display results
MsgBox "The cost of a new purchase is: " & newPurchase
MsgBox "The running cost is: " & allPurchase
End Sub

```

The above procedure begins with declaring a static variable named `allPurchase` and two other local variables: `newPurchase` and `purchCost`. The `InputBox` function used in this procedure displays a dialog box and waits for the user to enter the value. As soon as you input the value and click OK, Visual Basic assigns this value to the variable `newPurchase`.

The `InputBox` function is discussed in detail in Chapter 5. Because the result of the `InputBox` function is always a string, the `newPurchase` variable was declared as the String data type. You can't, however, use strings in mathematical calculations. That's why the next instruction uses a type conversion function (`CSng`) to translate the text value into a numeric variable of the Single data type. The `CSng` function requires one argument — the value you want to translate. To find out more about the `CSng` function, position the insertion point anywhere within the word `CSng` and press F1. The number obtained as the result of the `CSng` function is then stored in the variable `purchCost`.

The next instruction, `allPurchase = allPurchase + purchCost`, adds to the current purchase value the new value supplied by the `InputBox` function.

2. Position the cursor anywhere within the `CostOfPurchase` procedure and press F5. When the dialog box appears, enter a number. For example, enter **100** and click **OK** or press **Enter**. Visual Basic displays the message “The cost of a new purchase is: 100.” Click **OK** in the message box. Visual Basic displays the second message “The running cost is: 100.”
3. When you run this procedure for the first time, the content of the `allPurchase` variable is the same as the content of the `purchCost` variable.
4. Rerun the same procedure. When the input dialog appears, enter another number. For example, enter **50** and click **OK** or press **Enter**. Visual Basic displays the message “The cost of a new purchase is: 50.” Click **OK** in the message box. Visual Basic displays the second message “The running cost is: 150.”

When you run the procedure the second time, the value of the static variable is increased by the new value supplied in the dialog box. You can run the `CostOfPurchase` procedure as many times as you want. The `allPurchase` variable will keep the running total for as long as the project is open.



## Declaring and Using Object Variables

The variables that you've learned about thus far are used to store data. Storing data is the main reason for using "normal" variables in your procedures. In addition to the normal variables that store data, there are special variables that refer to the Visual Basic objects. These variables are called *object variables*. In Chapter 3, you worked with several objects in the Immediate window. Now you will learn how you can represent an object with the object variable.

Object variables don't store data; instead, they tell where the data is located. For example, with the object variable you can tell Visual Basic that the data is located in cell E10 of the active worksheet. Object variables make it easy to locate data. When writing Visual Basic procedures, you often need to write long instructions, such as:

```
Worksheets("Sheet1").Range(Cells(1, 1), Cells(10, 5)).Select
```

Instead of using long references to the object, you can declare an object variable that will tell Visual Basic where the data is located. Object variables are declared similarly to the variables you already know. The only difference is that after the `As` keyword, you enter the word `Object` as the data type. For instance:

```
Dim myRange As Object
```

The statement above declares the object variable named `myRange`.

Well, it's not enough to declare the object variable. You also have to assign a specific value to the object variable before you can use this variable in your procedure. Assign a value to the object variable by using the `Set` keyword. The `Set` keyword must be followed by the equals sign and the value that the variable will refer to. For example:

```
Set myRange = Worksheets("Sheet1").Range(Cells(1, 1), Cells(10, 5))
```

The above statement assigns a value to the object variable `myRange`. This value refers to cells A1:E10 in Sheet1. If you omit the word `Set`, Visual Basic will display an error message — "Run-time error 91: Object variable or With block variable not set."

Again, it's time to see a practical example. The `UseObjVariable` procedure shown in Hands-On 4.5 demonstrates the use of the object variable called `myRange`.



### Hands-On 4.5. Writing a VBA Procedure with Object Variables

1. In the Code window of the Variables module, write the following procedure:

```
Sub UseObjVariable()  
    Dim myRange As Object  
    Set myRange = Worksheets("Sheet1").Range(Cells(1, 1), Cells(10, 5))  
    myRange.BorderAround Weight:=xlMedium  
  
    With myRange.Interior  
        .ColorIndex = 6  
        .Pattern = xlSolid  
    End With  
  
    Set myRange = Worksheets("Sheet1").Range(Cells(12, 5), Cells(12, 10))  
    myRange.Value = 54  
  
    Debug.Print IsObject(myRange)  
End Sub
```

Let's examine the code of the UseObjVariable procedure line by line. The procedure begins with the declaration of the object variable `myRange`. The next statement sets the object variable `myRange` to the range A1:E10 on Sheet1. From now on, every time you want to reference this range, instead of using the entire object's address, you'll use the shortcut — the name of the object variable. The purpose of this procedure is to create a border around the range A1:E10. Instead of writing a long instruction:

```
Worksheets("Sheet1").Range(Cells(1, 1), Cells(10, 5)).BorderAround  
    Weight:=xlMedium
```

you can take a shortcut by using the name of the object variable:

```
myRange.BorderAround Weight:=xlMedium
```

The next series of statements changes the color of the selected range of cells (A1:E10). Again, you don't need to write the long instruction to reference the object that you want to manipulate. Instead of the full object name, you can use the `myRange` object variable. The next statement assigns a new reference to the object variable `myRange`. Visual Basic forgets the old reference, and the next time you use `myRange`, it refers to another range (E12:J12).

After the number 54 is entered in the new range (E12:J12), the procedure shows you how you can make sure that a specific variable is of the Object type. The instruction `Debug.Print IsObject(myRange)` will enter True in the Immediate window if `myRange` is an object variable. `IsObject` is a VBA function that indicates whether a specific value represents an object variable.

2. Position the cursor anywhere within the UseObjVariable procedure and press F5.

## Advantages of Using Object Variables

- They can be used instead of the actual object.
- They are shorter and easier to remember than the actual values to which they point.
- You can change their meaning while your procedure is running.

## Using Specific Object Variables

The object variable can refer to any type of object. Because Visual Basic has many types of objects, it's a good idea to create object variables that refer to a particular type of object to make your programs more readable and faster. For instance, in the UseObjVariable procedure (see the previous section), instead of the generic object variable (`Object`), you can declare the `myRange` object variable as a `Range` object:

```
Dim myRange As Range
```

If you want to refer to a particular worksheet, then you can declare the `Worksheet` object:

```
Dim mySheet As Worksheet  
Set mySheet = Worksheets("Marketing")
```

When the object variable is no longer needed, you can assign `Nothing` to it. This frees up memory and system resources:

```
Set mySheet = Nothing
```

## Finding a Variable Definition

When you find an instruction in a VBA procedure that assigns a value to a variable, you can quickly locate the definition of the variable by selecting the variable name and pressing Shift-F2 or choosing View | Definition. Visual Basic will jump to the variable declaration line. Press Ctrl-Shift-F2 or choose View | Last Position to return your mouse pointer to its previous position. Let's take a look at how this works in Hands-On 4.6.



### Hands-On 4.6. Finding a Variable Definition in the Code Window

1. Locate the code of the `CostOfPurchase` procedure.
2. Locate the statement `purchCost = CSng(newPurchase)`.

3. Right-click the variable name and choose **Definition** from the shortcut menu.
4. Return to the previous location by pressing **Ctrl-Shift-F2**.
5. Try finding definitions of other variables in other procedures created in this chapter using both methods of jumping to the variable definition.

### Determining a Data Type of a Variable

You can find out the type of a variable by using one of the VBA built-in functions. The `VarType` function returns an integer indicating the type of a variable. Figure 4.3 displays the `VarType` function's syntax and the values it returns.

Let's see how you can use the `VarType` function in the Immediate window.



### Hands-On 4.7. Using the Built-in VarType Function

1. In the Visual Basic Editor window, choose **View | Immediate Window**.
2. Type the following statements that assign values to variables:

```
age = 18
birthdate = #1/1/1981#
firstName = "John"
```

3. Now ask Visual Basic what type of data each of the variables holds:

```
?VarType (age)
```

When you press **Enter**, Visual Basic returns 2. As shown in Figure 4.3, the number 2 represents the Integer data type. If you type:

```
?VarType (birthdate)
```

Visual Basic returns 7 for Date. If you make a mistake in the variable name (let's say you type `birthday`, instead of `birthdate`), Visual Basic returns zero (0). If you type:

```
?VarType (firstName)
```

Visual Basic tells you that the value stored in the variable `firstName` is a String type (8).

The `Select Case` statement, which is covered in Chapter 6, can be used to write a function procedure (functions are covered in the next chapter) to return a user-friendly name of the variable type.

Excel Developer Home &gt; Visual Basic for Applications Language Reference &gt; Visual Basic Language Reference &gt; Functions



Search help

More on Office.com: [downloads](#) | [images](#) | [templates](#)

## VarType Function

Returns an **Integer** indicating the subtype of a variable.

### Syntax

**VarType**(*varname*)

The required *varname* argument is a Variant containing any variable except a variable of a user-defined type.

### Return Values

Constant	Value	Description
<b>vbEmpty</b>	0	Empty (uninitialized)
<b>vbNull</b>	1	Null (no valid data)
<b>vbInteger</b>	2	Integer
<b>vbLong</b>	3	Long integer
<b>vbSingle</b>	4	Single-precision floating-point number
<b>vbDouble</b>	5	Double-precision floating-point number
<b>vbCurrency</b>	6	Currency value
<b>vbDate</b>	7	Date value
<b>vbString</b>	8	String
<b>vbObject</b>	9	Object
<b>vbError</b>	10	Error value
<b>vbBoolean</b>	11	Boolean value
<b>vbVariant</b>	12	<b>Variant</b> (used only with arrays of variants)
<b>vbDataObject</b>	13	A data access object
<b>vbDecimal</b>	14	Decimal value
<b>vbByte</b>	17	Byte value
<b>vbLongLong</b>	20	LongLong integer (Valid on 64-bit platforms only.)
<b>vbUserDefinedType</b>	36	Variants that contain user-defined types
<b>vbArray</b>	8192	Array

**FIGURE 4.3.** With the built-in VarType function, you can learn the data type the variable holds.

```

Public Function GetFriendlyVarType(vType As Variant) As String
    Select Case VarType(vType)
        Case 0
            GetFriendlyVarType = "Empty"
        Case 1
            GetFriendlyVarType = "Null"
        Case 2
            GetFriendlyVarType = "Integer"
        Case 3
            GetFriendlyVarType = "Long"
        Case 4
            GetFriendlyVarType = "Single"
        Case 5
            GetFriendlyVarType = "Double"
        Case 6
            GetFriendlyVarType = "Currency"
        Case 7
            GetFriendlyVarType = "Date"
        Case 8
            GetFriendlyVarType = "String"
        Case 9
            GetFriendlyVarType = "Object"
        Case 10
            GetFriendlyVarType = "Error"
        Case 11
            GetFriendlyVarType = "Boolean"
        Case 12
            GetFriendlyVarType = "Variant"
        Case 13
            GetFriendlyVarType = "Data Object"
        Case 14
            GetFriendlyVarType = "Decimal"
        Case 17
            GetFriendlyVarType = "Byte"
        Case 20
            GetFriendlyVarType = "LongLong"
        Case 36
            GetFriendlyVarType = "Variant containing UDT"
        Case Is >= 8192
            GetFriendlyVarType = "Array"
        Case Else
            GetFriendlyVarType = "Unknown"
    End Select
End Function

```

To test the above function, a value was assigned to `myVariable` in the Immediate window. The result of the `GetFriendlyVarType` function was assigned to the variable named `result`. Then, a question mark was used to query the contents of the `result` variable:

```

myVariable="Today is Saturday"
result = GetFriendlyVarType(myVariable)
?result

```

```
myVariable=#07-12-2010#  
result = GetFriendlyVarType(myVariable)  
?result
```

The first variable assignment returned String as the data type, and the second one returned Date.

## USING CONSTANTS IN VBA PROCEDURES

---

The contents of a variable can change while your procedure is executing. If your procedure needs to refer to unchanged values over and over again, you should use constants. A *constant* is like a named variable that always refers to the same value. Visual Basic requires that you declare constants before you use them. Declare constants by using the `Const` statement, as in the following examples:

```
Const dialogName = "Enter Data" As String  
Const sIsTax = 8.5  
Const ColorIdx = 3
```

A constant, like a variable, has a scope. To make a constant available within a single procedure, declare it at the procedure level, just below the name of the procedure. For instance:

```
Sub WedAnniv()  
    Const Age As Integer = 25  
    MsgBox (Age)  
End Sub
```

If you want to use a constant in all the procedures of a module, use the `Private` keyword in front of the `Const` statement. For instance:

```
Private Const disk As String = "A:"
```

The `Private` constant has to be declared at the top of the module, just before the first `Sub` statement.

If you want to make a constant available to all modules in the workbook, use the `Public` keyword in front of the `Const` statement. For instance:

```
Public Const NumOfChars As Integer = 255
```

The `Public` constant has to be declared at the top of the module, just before the first `Sub` statement.

When declaring a constant, you can use any one of the following data types: Boolean, Byte, Integer, Long, Currency, Single, Double, Date, String, or Variant.

Like variables, several constants can be declared on one line if separated by commas. For instance:

```
Const Age As Integer = 25, City As String = "Denver"
```

Using constants makes your VBA procedures more readable and easier to maintain. For example, if you refer to a certain value several times in your procedure, use a constant instead of the value. This way, if the value changes (for example, the sales tax goes up), you can simply change the value in the declaration of the `Const` statement instead of tracking down every occurrence of that value.

### Built-in Constants

Both Microsoft Excel and Visual Basic for Applications have a long list of predefined constants that do not need to be declared. These built-in constants can be looked up using the Object Browser window. Let's proceed to Hands-On 4.8 where we open the Object Browser to take a look at the list of Excel constants.



#### Hands-On 4.8. Viewing Excel Constants in the Object Browser

1. In the Visual Basic Editor window, choose **View | Object Browser**.
2. In the Project/Library list box, click the drop-down arrow and select **Excel**.
3. Enter **constants** as the search text in the Search box and press **Enter** or click the **Search** button. Visual Basic shows the result of the search in the Search Results area.
4. Scroll down in the Classes list box to locate and then select **Constants** as shown in Figure 4.4. The right side of the Object Browser window displays a list of all built-in constants that are available in the Microsoft Excel object library. Notice that the names of all the constants begin with the prefix "xl."
5. To look up VBA constants, choose **VBA** in the Project/Library list box. Notice that the names of the VBA built-in constants begin with the prefix "vb."

The best way to learn about predefined constants is by using the macro recorder. Let's take a few minutes in Hands-On 4.9 to record the process of minimizing the active window.



#### Hands-On 4.9. Learning about Constants Using the Built-in Macro Recorder

1. In the Microsoft Excel window, choose **View | Macros | Record Macro**.
2. Type **MiniWindow** as the name of the macro. Under Store macro in, select **This Workbook**. Then click **OK**.



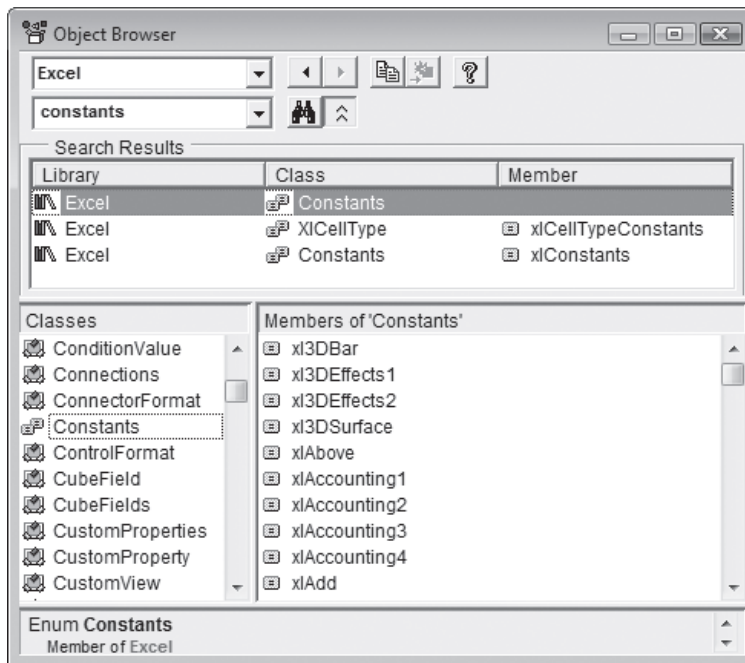
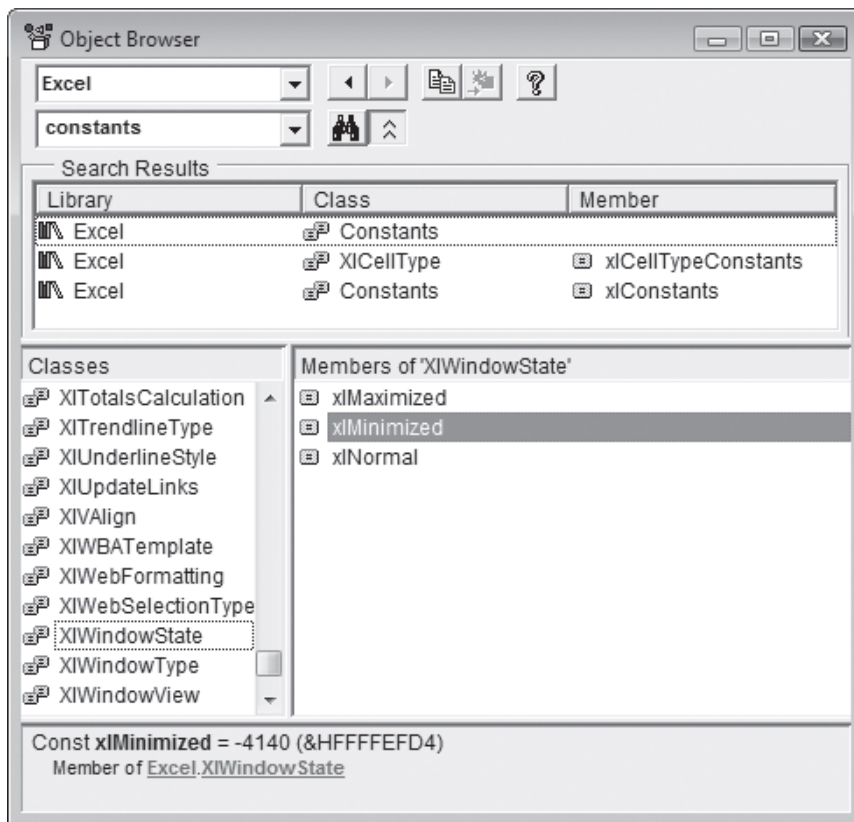


FIGURE 4.4. Use the Object Browser to look up any built-in constant.

3. Click the **Minimize** button. Make sure you minimize the document window and not the Excel application window.
4. Choose **View | Macros | Stop Recording**.
5. Maximize the minimized document window.
6. Switch to the Visual Basic Editor window and double-click **Module1** in the Project Explorer window. The Code window displays the following procedure:

```
Sub MiniWindow()
'
' MiniWindow Macro
'
ActiveWindow.WindowState = xlMinimized
End Sub
```

Sometimes you may see VBA procedures that use values instead of built-in constant names. For example, the actual value of `xlMaximized` is `-4137`, `xlMinimized` has a value of `-4140`, and `xlNormal` has a value of `-4143` as shown in Figure 4.5.



**FIGURE 4.5.** You can see the actual value of a constant by selecting its name in the Object Browser.

## CHAPTER SUMMARY

This chapter introduced several new VBA concepts such as data types, variables, and constants. You learned how to declare various types of variables and define their types. You also saw the difference between a variable and a constant. Now that you know what variables are and how to use them, you are capable of creating VBA procedures that can manipulate data in more meaningful ways than you saw in previous chapters.

In the next chapter, you will expand your VBA knowledge by learning how to write procedures with arguments as well as function procedures. In addition, you will learn about built-in functions that will allow your VBA procedure to interact with users.