Understanding Program Variables

Your VBA procedures often need to store temporary values for use in statements and calculations that come later in the code. For example, you might want to store values for total sales and total expenses to use later in a gross margin calculation. Although you probably could get away with using the underlying application to store these values (in, say, a cell in an Excel worksheet), this almost always isn’t very practical. Instead, VBA (like all programming languages) lets you store temporary values in special memory locations called variables. This chapter explains this important topic and shows you how to use variables in your VBA procedures.

Declaring Variables

Declaring a variable tells VBA the name of the variable you’re going to use. (It also serves to specify the data type of the variable, which I’ll explain later in this chapter.) Note that at this point you’re not assigning a value to the variable. That comes later. All you’re doing now is telling VBA that the variable exists. You declare variables by including Dim statements (Dim is short for dimension) at the beginning of each Sub or Function procedure.

NOTE

Technically, you can put variable declarations anywhere you like within a procedure and VBA won’t complain. The only real restriction is that the Dim statement must precede the first use of the variable in a procedure. Having said all that, however, it’s not only traditional, but also clearer, to list all your Dim statements together at the top of a procedure.
In its simplest form, a `Dim` statement has the following syntax:

```vba
Dim variableName
```

Here, `variableName` is the name of the variable. You make up these names yourself, but you need to bear a few restrictions in mind:

- The name must begin with a letter.
- The name can't be longer than 255 characters.
- The name can't be a VBA keyword (such as `Dim`, `Sub`, or `End`).
- The name can't contain a space or any of the following characters: . ! # $ % & @.

For example, the following statement declares a variable named `totalSales`:

```vba
Dim totalSales
```

To avoid confusing variable names with the names of things that are built into the VBA language, many programmers begin their variable names with a lowercase letter. If the name contains multiple "words," then each subsequent word should use an uppercase first letter (for example, `totalSales` or `newFileName`). This is the style I use in this book. (Programming types call it `camel style`, thanks to the "humps" created by the uppercase letters.)

Also, note that VBA preserves the case of your variable names throughout a procedure. For example, if you declare a variable named `totalSales` and you later enter this variable name as, say, `totalsales`, VBA will convert the name to `totalSales` automatically as part of its syntax checking. This means two things:

- If you want to change the case used in a variable, change the first instance of the variable (usually the `Dim` statement).
- After you've declared a variable, you should enter all subsequent references to the variable entirely in lowercase. Not only is this easier to type, but you'll immediately know whether you've misspelled the variable name if you see that VBA doesn't change the case of the variable name after you enter the line.

Most programmers set up a declaration section at the beginning of each procedure and use it to hold all their `Dim` statements. Then, after the variables have been declared, you can use them throughout the procedure. Listing 3.1 shows a `Function` procedure that declares two variables—`totalSales` and `totalExpenses`—and then uses Excel's `Sum` function to store a range sum in each variable. Finally, the `GrossMargin` calculation uses each variable to return the function result.
Avoiding Variable Errors

Listing 3.1  A Function That Uses Variables to Store the Intermediate Values of a Calculation

Function GrossMargin()
    ' Declarations
    Dim totalSales
    Dim totalExpenses
    ' Code
    totalSales = Application.Sum(Range("Sales"))
    totalExpenses = Application.Sum(Range("Expenses"))
    GrossMargin = (totalSales - totalExpenses) / totalSales
End Function

In the GrossMargin function, notice that you store a value in a variable with a simple assignment statement of the following form:

variableName = value

To conserve space, you can declare multiple variables on a single line. In the GrossMargin function, for example, you could declare totalSales and totalExpenses using the following statement:

Dim totalSales, totalExpenses

Listing 3.1 gets its values from the Excel worksheet by using the Range method. For the details, see “Using the Range Method,” p. 153.

Avoiding Variable Errors

One of the most common errors in VBA procedures is to declare a variable and then later misspell the name. For example, suppose I had entered the following statement in the GrossMargin procedure from Listing 3.1:

totlExpenses = Application.Sum(Range("Expenses"))

Here, totlExpenses is a misspelling of the variable named totalExpenses. VBA supports implicit declarations, which means that if it sees a name it doesn’t recognize, it assumes that the name belongs to a new variable. In this case, VBA would assume that totlExpenses is a new variable, proceed normally, and calculate the wrong answer for the function.

To avoid this problem, you can tell VBA to generate an error whenever it comes across a name that hasn’t been declared explicitly with a Dim statement. There are two ways to do this:
For an individual module, enter the following statement at the top of the module:

```
Option Explicit
```

To force VBA to add this statement automatically to all your modules, in the Visual Basic Editor select Tools, Options, display the Editor tab in the Options dialog box that appears, and activate the Require Variable Declaration check box.

**NOTE**

Activating the Require Variable Declaration check box forces VBA to add the `Option Explicit` statement at the beginning of each new module. However, it doesn't add this statement to any existing modules; you need to do that by hand.

---

**Variable Data Types**

The *data type* of a variable determines the kind of data the variable can hold. You specify a data type by including the `As` keyword in a `Dim` statement. Here is the general syntax:

```
Dim variableName As DataType
```

`variableName` is the name of the variable and `DataType` is one of the data types. Here's a rundown of the most useful VBA data types:

- **String**—This type holds *strings*, which are simple text values. Here's a sample declaration and assignment statement (note the use of quotation marks in the assignment statement value; this tells VBA that the value is a string):

  ```
  Dim newFileName As String
  newFileName = "Budget Notes.doc"
  ```

- **Date**—This type holds *date* values, which refers to dates and/or times. Here are a few examples (note the use of the `#` character around the values; this tells VBA that the values are dates and/or times):

  ```
  Dim myBirthDate As Date
  Dim myBirthTime As Date
  Dim anotherDate As Date
  myBirthDate = #8/23/59#
  myBirthTime = #3:02 AM#
  anotherDate = #4/27/07 16:05#
  ```

- **Object**—You use this type to hold generic *objects*, which I discuss in detail in Chapter 5, “Working with Objects.”

- **Byte**—This rarely used type holds small, positive integer values (from 0 to 255).

- **Integer**—This type holds *integer* values, which VBA defines as whole numbers between –32,768 and 32,767. Here’s an example:

  ```
  Dim paragraphNumber As Integer
  paragraphNumber = 1
  ```
Variable Data Types

- **Long**—This type holds *long integer* values, which VBA defines as whole numbers between –2,147,483,648 and 2,147,483,647. Here’s an example (note that you don’t include commas—or periods, if you’re in Europe—in numbers that would normally use one or more thousands separators):

  ```vba
  Dim wordCount As Long
  wordCount = 100000
  ```

- **Boolean**—This type holds *Boolean* values, which take one of two values: True or False. Here’s an example:

  ```vba
  Dim documentSaved As Boolean
  documentSaved = False
  ```

- **Currency**—This type holds *monetary* values. The value range is from –922,337,203,685,477.5808 to 922,337,203,685,477.5807.

- **Single**—This type holds *single-precision floating point* values, which are numbers that have a decimal component. Here’s an example:

  ```vba
  Dim averageUnitSales As Single
  averageUnitSales = 50.3
  ```

- **Double**—This type holds *double-precision floating point* values, which can accommodate much larger or smaller numbers than the Single type. Note, however, that the range available with the Single type should be more than enough for your VBA macros, so you’ll probably never use the **Double** type. Here’s an example:

  ```vba
  Dim atomsInTheUniverse As Double
  atomsInTheUniverse = 2.0E+79
  ```

**NOTE**

Double values often use *exponential notation*, such as the value 2.0E+79 used in the **Double** example. A positive number, say X, after the E symbol means that you move the decimal point X positions to the right to get the actual number. So, for example, 2.0E+3 is the same thing as 2000. A negative number, say –X, after the E means that you move the decimal point X positions to the left. So 3.14E-4 is the equivalent of 0.000314.

Here are a few notes to keep in mind when using data types:

- If you don’t include a data type when declaring a variable, VBA assigns the **Variant** data type. This enables you to store any kind of data in the variable. However, this isn’t a good idea because Variant variables use more memory and are much slower than the other data types. Therefore, always give your variables a specific data type. Note, however, that you may on occasion need a variable that can assume different data types. In that case, you should declare the variable using the **Variant** type.

  ➔ For an example of a situation in which declaring a variable as a **Variant** is a good idea, see “Getting Input Using InputBox,” p. 50.
If you declare a variable to be one data type and then try to store a value of a different data type in the variable, VBA often displays an error. For example, if you declare a variable using the `Single` type and you try to assign a value that’s outside the `Single` type’s allowable range, VBA displays an “Overflow” error message when you attempt to run the procedure.

To specify the data type of a procedure argument, use the `As` keyword in the argument list. For example, the following `Function` statement declares variables `Sales` and `Expenses` to be `Currency`:

```vba
Function GrossMargin(Sales As Currency, Expenses As Currency)
```

To specify the data type of the return value for a `Function` procedure, use the `As` keyword at the end of the `Function` statement:

```vba
Function GrossMargin(Sales, Expenses) As Single
```

Many programmers remind themselves of each variable’s data type by applying data type prefixes to the variable names. For example, the data type prefix for a `String` variable is `str`, so the declaration for such a variable might look like this:

```vba
Dim strName As String
```

This helps you avoid programming errors because you’re less likely to try and store, say, an `Integer` value in a `String` variable if that variable’s name begins with `str`. Here are some other common data type prefixes:

<table>
<thead>
<tr>
<th>Data Type</th>
<th>Prefix</th>
</tr>
</thead>
<tbody>
<tr>
<td>String</td>
<td>str or s</td>
</tr>
<tr>
<td>Date</td>
<td>dte or dtm</td>
</tr>
<tr>
<td>Object</td>
<td>obj</td>
</tr>
<tr>
<td>Byte</td>
<td>byt</td>
</tr>
<tr>
<td>Integer</td>
<td>int or i</td>
</tr>
<tr>
<td>Long</td>
<td>lng</td>
</tr>
<tr>
<td>Boolean</td>
<td>bln or b</td>
</tr>
<tr>
<td>Currency</td>
<td>cur</td>
</tr>
<tr>
<td>Single</td>
<td>sgl or sng</td>
</tr>
<tr>
<td>Double</td>
<td>dbl</td>
</tr>
<tr>
<td>Variant</td>
<td>vnt</td>
</tr>
</tbody>
</table>
Variable Data Types

Changing the Default Data Type

I mentioned in the preceding section that VBA assigns the Variant type to a variable if you don’t specify a data type. However, VBA supports a number of DefType statements that let you redefine the default data type. These statements all use the following syntax:

\[
\text{DefType} \ letter1[\text{-} letter2]
\]

Here, Type is a three- or four-letter code that specifies the data type, and letter1 and letter2 define a range of letters. Note that this is a module-level statement, so you must place it at the top of a module, before any procedures or functions.

The idea is that any variable (or function argument or function result) that begins with one of these letters will be assigned the specified data type by default. For example, the DefInt keyword is used to set the default data type to Integer. If you want VBA to assign, say, the Integer data type to any variables that begin with the letters X through Z, you would add the following statement at the module level:

\[
\text{DefInt} \ X-Z
\]

Table 3.1 lists the various DefType keywords and the data types they represent.

<table>
<thead>
<tr>
<th>DefType</th>
<th>Data Type</th>
</tr>
</thead>
<tbody>
<tr>
<td>DefBool</td>
<td>Boolean</td>
</tr>
<tr>
<td>DefByte</td>
<td>Byte</td>
</tr>
<tr>
<td>DefInt</td>
<td>Integer</td>
</tr>
<tr>
<td>DefLng</td>
<td>Long</td>
</tr>
<tr>
<td>DefCur</td>
<td>Currency</td>
</tr>
<tr>
<td>DefSng</td>
<td>Single</td>
</tr>
<tr>
<td>DefDbl</td>
<td>Double</td>
</tr>
<tr>
<td>DefDate</td>
<td>Date</td>
</tr>
<tr>
<td>DefStr</td>
<td>String</td>
</tr>
<tr>
<td>DefObj</td>
<td>Object</td>
</tr>
<tr>
<td>DefVar</td>
<td>Variant</td>
</tr>
</tbody>
</table>

Creating User-Defined Data Types

VBA’s built-in data types cover a lot of ground and should be sufficient to meet most of your needs. However, VBA also lets you set up user-defined data types. These are handy for storing similar types of data in a single structure. For example, suppose your program is
working with car makes and models. In this case, you might need to work with values for the manufacturer, the model, the year the car was made, and the purchase price. One way to go about this would be to set up variables for each item of data, like so:

```vba
Dim carMake As String  
Dim carModel As String  
Dim yearMade As Integer  
Dim carPrice As Currency  
```

This approach works, but what if you need to work with the data from multiple cars at once? You could set up new variables for each car, but that seems too inefficient. A better way is to define a "CarInfo" data type that holds all the required information. Here's how you would do it:

```vba
Type CarInfo  
  make As String  
  model As String  
  made As Integer  
  price As Currency  
End Type
```

The `Type` keyword tells VBA that you're creating a user-defined data type. In this example, the new data type is named `CarInfo`. The statements between `Type` and `End Type` define the various elements within the new data type. Note that you need to place this definition at the module level; VBA doesn't let you define new data types within a procedure.

Now you use the data type as you would any other. For example, the following statement declares a new variable named `myCar` to be of type `CarInfo`:

```vba
Dim myCar As CarInfo
```

From here, you refer to the various elements within the data type by separating the variable name and the element name with a period (.), like so:

```vba
myCar.make = "Porsche"  
myCar.model = "911 Turbo"  
myCar.made = 2007  
myCar.price = 122000
```

### Using Array Variables

In VBA, an array is a group of variables of the same data type. Why would you need to use an array? Well, suppose you wanted to store twenty employee names in variables to use in a procedure. One way to do this would be to create 20 variables named, say, `employee1`, `employee2`, and so on. However, it's much more efficient to create a single `employees` array variable that can hold up to 20 names. VBA creates a single variable with 20 different "slots" into which you can add data (such as employee names). Such an array variable is akin to an Excel range that consists of 20 cells in a row or column: the range is a single entity, but it contains 20 slots (cells) into which you can insert data. The major difference is
that you almost always use an array variable to hold data of a single type, such as String. When you declare an array variable you specify the data type, as shown here:

```vba
Dim employees(19) As String
```

As you can see, this declaration is very similar to one you would use for a regular variable. The difference is the 19 enclosed in parentheses. The parentheses tell VBA that you're declaring an array, and the number tells VBA how many elements you'll need in the array. Why 19 instead of 20? Well, each element in the array is assigned a subscript, where the first element's subscript is 0, the second is 1, and so on up to, in this case, 19. Therefore, the total number of elements in this array is 20.

You use a subscript to refer to any element simply by enclosing its index number in the parentheses, like so:

```vba
employees(0) = "Ponsonby"
```

By default, the subscripts of VBA arrays start at 0 (this is called the lower bound of the array) and run up to the number you specify in the Dim statement (this is called the upper bound of the array). If you would prefer your array index numbers to start at 1, include the following statement at the top of the module (in other words, before declaring your first array and before your first procedure):

```vba
Option Base 1
```

Note, too, that after resetting the lower bound in this way, if you want to declare an array with the same number of elements, then you need to adjust the upper bound in the Dim statement accordingly. For example, with the lower bound set to 1, if you want to declare an array variable named `employees` and you want it to hold up to 20 names, then you need to declare it like so:

```vba
Dim employees(20) As String
```

### Dynamic Arrays

What do you do if you're not sure how many subscripts you'll need in an array? You could guess at the correct number, but that will almost always leave you with one of the following problems:

- If you guess too low and try to access a subscript higher than the array's upper bound, VBA will generate an error message.
- If you guess too high, VBA will still allocate memory to the unused portions of the array, so you'll waste precious system resources.

To avoid both of these problems, you can declare a *dynamic* array by leaving the parentheses blank in the Dim statement:

```vba
Dim myArray() As Double
```
Then, when you know the number of elements you need, you can use a `ReDim` statement to allocate the correct number of subscripts (notice that you don’t specify a data type in the `ReDim` statement):

```vba
ReDim myArray(52)
```

The following is a partial listing of a procedure named `PerformCalculations`. The procedure declares `calcValues` as a dynamic array and `totalValues` as an integer. Later in the procedure, `totalValues` is set to the result of a function procedure named `GetTotalValues`. The `ReDim` statement then uses `totalValues` to allocate the appropriate number of subscripts to the `calcValues` array.

```vba
Sub PerformCalculations()
    Dim calcValues() As Double, totalValues as Integer
    ...
    totalValues = GetTotalValues()
    ReDim calcValues(totalValues)
    ...
End Sub
```

The `ReDim` statement reinitializes the array so that any values stored in the array are lost. If you want to preserve an array’s existing values, use `ReDim` with the `Preserve` option, as follows:

```vba
ReDim Preserve myArray(52)
```

Listing 3.2 presents a more concrete example. (Note that this procedure uses lots of VBA code that you haven’t seen yet, so don’t be discouraged if you don’t fully understand what’s happening here.)

```vba
Sub StoreWorksheetNames()
    Dim sheetNames() As String
    Dim totalSheets As Integer
    Dim sheet As Worksheet
    Dim i As Integer
    Dim strMessage As String

    ' Store the total number of worksheets that are in the current workbook
    totalSheets = ActiveWorkbook.Worksheets.Count

    ' Now redimension the dynamic array
    ReDim sheetNames(totalSheets)

    ' Loop through the worksheets to store the names in the array
```

**NOTE**

Listing 3.2 presents a more concrete example. (Note that this procedure uses lots of VBA code that you haven’t seen yet, so don’t be discouraged if you don’t fully understand what’s happening here.)
For i = 1 To totalSheets
    sheetNames(i - 1) = ActiveWorkbook.Worksheets(i).Name
Next 'i
'
Loop through the array to add the names to a string
strMessage = "Here are the worksheet names:" & vbCrLf
For i = 0 To totalSheets - 1
    strMessage = strMessage & sheetNames(i) & vbCrLf
Next 'i
'
Display the worksheet names
MsgBox strMessage
End Sub

This procedure begins by declaring `sheetNames` as a dynamic array. It then uses the `totalSheets` variable to store the total number of worksheets that are in the current workbook. The procedure then sets the size of the array based on the `totalSheets` value:

```vba
ReDim sheetNames(totalSheets)
```

The procedure then uses one loop (see Chapter 6, “Controlling Your VBA Code”) to store the worksheet names in the array and a second loop to add the worksheet names to the `strMessage` variable, which is a `String` value. Finally, the procedure uses the `MsgBox` function to display the string, as shown in Figure 3.1.

![Figure 3.1](image)
The results of the dynamic array procedure in Listing 3.2.

If your program needs to know the lower bound and the upper bound of an array, VBA provides a couple of functions that can do the job:

- `LBound(arrayName)` Returns the lower bound of the array given by `arrayName`.
- `UBound(arrayName)` Returns the upper bound of the array given by `arrayName`. 
Multidimensional Arrays

If you enter a single number between the parentheses in an array’s Dim statement, VBA creates a one-dimensional array. But you also can create arrays with two or more dimensions (60 is the maximum). For example, suppose you wanted to store both a first name and a last name in your employee array. To store two sets of data with each element, you would declare a two-dimensional array, like so:

Dim employees(19,1) As String

The subscripts for the second number work like the subscripts you’ve seen already. In other words, they begin at 0 and run up to the number you specify. So this Dim statement sets up a “table” (or a matrix, as it’s usually called) with 20 “rows” (one for each employee) and two “columns” (one for the first name and one for the last name). So if a one-dimensional array is like an Excel range consisting of cells in a single row or column, a multidimensional array is like an Excel range consisting of cells in multiple rows or columns.

Here are two statements that initialize the data for the first employee:

employees(0,0) = "Biff"
employees(0,1) = "Ponsonby"

Working with Constants

Constants are values that don’t change. They can be numbers, strings, or other values, but, unlike variables, they keep their value throughout your code. VBA recognizes two types of constants: built-in and user-defined.

Using Built-In Constants

Many properties and methods have their own predefined constants. For Excel objects, these constants begin with the letters xl. For Word objects, the constants begin with wd. For VBA objects, the constants begin with vb.

For example, Excel’s Window object has a WindowState property that recognizes three built-in constants: xlNormal (to set a window in its normal state), xlMaximized (to maximize a window), and xlMinimized (to minimize a window). To maximize the active window, for example, you would use the following statement:

ActiveWindow.WindowState = xlMaximized

If you want to see a list of all the built-in constants for an application, open the Visual Basic Editor, choose View, Object Browser (or click F2), use the Project/Library list to click the application name (such as Word or Excel), and then click <globals> at the top of the Classes list. Scroll down the Members list until you get to the items that begin with the application’s constant prefix (xl for Excel, wd for Word, pp for PowerPoint, and ac for Access).
Creating User-Defined Constants

To create your own constants, use the Const statement:

```
Const CONSTANTNAME [As type] = expression
```

- **CONSTANTNAME**—The name of the constant. Most programmers use all-uppercase names for constants, which helps distinguish them from your regular variables as well as the VBA keywords.
- **As type**—Use this optional expression to assign a data type to the constant.
- **expression**—The value (or a formula that returns a value) that you want to use for the constant. You must use either a literal value or an expression that combines literal values and one or more other constants (as long as those constants have been declared before the current constant).

For example, the following statement creates a constant named DISCOUNT and assigns it the value 0.4:

```
Const DISCOUNT As Single = 0.4
```

Storing User Input in a Variable

Your VBA programs will usually be self-contained and run just fine on their own. However, you’ll likely come across situations where you’ll require some kind of custom input. For example, you might have a procedure that adjusts various aspects of a Word document. You could insert the name and location of a Word document into the procedure (this is called **hard-coding** the data), but that’s not very flexible if your procedure is capable of working with different documents. A better idea is to have your procedure prompt for the name and location of a document. Your procedure could then take that data and use it to work on the specified document.

Whatever type of input you ask for, the result needs to be stored in a variable so that the rest of your procedure can access it. The next couple of sections take you through some VBA techniques that enable you to prompt for data and then store that data in a variable.

Getting Input Using MsgBox

You’ve seen a couple of times already in this book that you can display information by using the MsgBox function. This is a very useful function, so let’s take a closer look at it. Here is the full syntax of this function:

```
MsgBox(Prompt[, Buttons][, Title][, HelpFile][, Context])
```

- **Prompt**
  - The message you want to display in the dialog box. (You can enter a string up to 1,024 characters long.)

- **Buttons**
  - A number or constant that specifies, among other things, the command buttons that appear in the dialog box. (See the next section.) The default value is 0.
Chapter 3 Understanding Program Variables

Title
The text that appears in the dialog box title bar. If you omit the title, VBA uses the name of the current program (for example, Microsoft Excel).

HelpFile
The text that specifies the Help file that contains the custom help topic. (I don’t discuss custom help topics in this book.) If you enter HelpFile, you also have to include Context. If you include HelpFile, a Help button appears in the dialog box.

Context
A number that identifies the help topic in HelpFile.

NOTE
There are a number of tutorials online that show you how to create a Help file. Type “creating help files” into your favorite search engine.

For example, the following statement displays the message dialog box shown in Figure 3.2:

MsgBox "You must enter a number between 1 and 100!", "Warning"

Figure 3.2
A simple message dialog box produced by the MsgBox function.

NOTE
The MsgBox function, like all VBA functions, needs parentheses around its arguments only when you use the function’s return value. See the section later in this chapter called “Getting Return Values from the Message Dialog Box” to learn about the return values produced by the MsgBox function.

TIP
For long prompts, VBA wraps the text inside the dialog box. If you’d prefer to create your own line breaks, use VBAs vbCrLf constant to insert a carriage-return and line-feed between each line:

MsgBox "First line" & vbCrLf & "Second line"

Setting the Style of the Message
The default message dialog box displays only an OK button. You can include other buttons and icons in the dialog box by using different values for the Buttons parameter. Table 3.2 lists the available options.
### Table 3.2 The MsgBox Buttons Parameter Options

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Buttons</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vbOKOnly</td>
<td>0</td>
<td>Displays only an OK button. (This is the default.)</td>
</tr>
<tr>
<td>vbOKCancel</td>
<td>1</td>
<td>Displays the OK and Cancel buttons.</td>
</tr>
<tr>
<td>vbAbortRetryIgnore</td>
<td>2</td>
<td>Displays the Abort, Retry, and Ignore buttons.</td>
</tr>
<tr>
<td>vbYesNoCancel</td>
<td>3</td>
<td>Displays the Yes, No, and Cancel buttons.</td>
</tr>
<tr>
<td>vbYesNo</td>
<td>4</td>
<td>Displays the Yes and No buttons.</td>
</tr>
<tr>
<td>vbRetryCancel</td>
<td>5</td>
<td>Displays the Retry and Cancel buttons.</td>
</tr>
<tr>
<td><strong>Icons</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vbCritical</td>
<td>16</td>
<td>Displays the Critical Message icon.</td>
</tr>
<tr>
<td>vbQuestion</td>
<td>32</td>
<td>Displays the Warning Query icon.</td>
</tr>
<tr>
<td>vbExclamation</td>
<td>48</td>
<td>Displays the Warning Message icon.</td>
</tr>
<tr>
<td>vbInformation</td>
<td>64</td>
<td>Displays the Information Message icon.</td>
</tr>
<tr>
<td><strong>Default Button</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vbDefaultButton1</td>
<td>0</td>
<td>The first button is the default (that is, the button selected when the user presses Enter).</td>
</tr>
<tr>
<td>vbDefaultButton2</td>
<td>256</td>
<td>The second button is the default.</td>
</tr>
<tr>
<td>vbDefaultButton3</td>
<td>512</td>
<td>The third button is the default.</td>
</tr>
<tr>
<td><strong>Modality</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>vbApplicationModal</td>
<td>0</td>
<td>The user must respond to the message box before continuing work in the current application.</td>
</tr>
<tr>
<td>vbSystemModal</td>
<td>4096</td>
<td>All applications are suspended until the user responds to the message box.</td>
</tr>
</tbody>
</table>

You derive the *Buttons* argument in one of two ways:

- By adding up the values for each option. For example, if you want the OK and Cancel buttons (value 1) and the Warning Message icon (value 48), then you specify the value 49.
- By using the VBA constants separated by plus signs (+). This is the better way to go because it makes your code much easier to read.
For example, Listing 3.3 shows a procedure named ButtonTest, and Figure 3.3 shows the resulting dialog box. Here, three variables—msgPrompt, msgButtons, and msgTitle—store the values for the MsgBox function's Prompt, Buttons, and Title arguments, respectively. In particular, the following statement derives the Buttons argument:

```vbnet
msgButtons = vbYesNo + vbQuestion + vbDefaultButton2
```

You also could derive the Buttons argument by adding up the values that these constants represent (4, 32, and 256, respectively), but the procedure becomes less readable that way.

### Listing 3.3 A Procedure That Creates a Message Dialog Box

```vbnet
Sub ButtonTest()
    Dim msgPrompt As String, msgTitle As String
    Dim msgButtons As Integer, msgResult As Integer

    msgPrompt = "Are you sure you want to display " & vbCrLf & _
                "the worksheet names?"
    msgButtons = vbYesNo + vbQuestion + vbDefaultButton2
    msgTitle = "Display Worksheet Names"

    msgResult = MsgBox(msgPrompt, msgButtons, msgTitle)
End Sub
```

### Figure 3.3
The dialog box that's displayed when you run the code in Listing 3.3.

---

**Getting Return Values from the Message Dialog Box**

A message dialog box that displays only an OK button is straightforward. The user either clicks OK or presses Enter to remove the dialog from the screen. The multibutton styles are a little different, however; the user has a choice of buttons to select, and your procedure should have a way to find out which button the user chose.

You do this by storing the MsgBox function's return value in a variable. Table 3.3 lists the seven possible return values.
Storing User Input in a Variable

Table 3.3 The MsgBox Function’s Return Values

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Button Selected</th>
</tr>
</thead>
<tbody>
<tr>
<td>vbOK</td>
<td>1</td>
<td>OK</td>
</tr>
<tr>
<td>vbCancel</td>
<td>2</td>
<td>Cancel</td>
</tr>
<tr>
<td>vbAbort</td>
<td>3</td>
<td>Abort</td>
</tr>
<tr>
<td>vbRetry</td>
<td>4</td>
<td>Retry</td>
</tr>
<tr>
<td>vbIgnore</td>
<td>5</td>
<td>Ignore</td>
</tr>
<tr>
<td>vbYes</td>
<td>6</td>
<td>Yes</td>
</tr>
<tr>
<td>vbNo</td>
<td>7</td>
<td>No</td>
</tr>
</tbody>
</table>

To process the return value, you test the value in the variable and have your procedure take appropriate action. You learn how to do this in Chapter 6. Listing 3.4 shows a revised version of ButtonTest that uses an If statement to see whether the msgResult value equals vbYes. If so, it means the user clicked Yes in the dialog box, so the procedure runs the StoreWorksheetNames procedure (see Listing 3.4); otherwise, it does nothing.

➔ To learn about the If statement, see “Using If...Then to Make True/False Decisions,” p. 92.
➔ For MsgBox functions that use three buttons, you need to use the Select Case statement to process the result; see “Using the Select Case Statement,” p. 97.

Listing 3.4  A Procedure that Handles the Return Value of the MsgBox Function

```vba
Sub ButtonTest2()
    Dim msgPrompt As String, msgTitle As String
    Dim msgButtons As Integer, msgResult As Integer

    msgPrompt = "Are you sure you want to display " & vbCrLf & _
                "the worksheet names?"
    msgButtons = vbYesNo + vbQuestion + vbDefaultButton2
    msgTitle = "Display Worksheet Names"

    msgResult = MsgBox(msgPrompt, msgButtons, msgTitle)
    If msgResult = vbYes Then
        StoreWorksheetNames
    End If
End Sub
```
Getting Input Using InputBox

As you’ve seen, the MsgBox function lets your procedures interact with the user and get some feedback. Unfortunately, this method limits you to simple command-button responses. For more varied user input, you need to use a more sophisticated technique. The rest of this chapter shows you just such a method: prompting the user for input using the InputBox function.

The InputBox function displays a dialog box with a message that prompts the user to enter data, and it provides a text box for the data itself. Here’s the syntax for this function:

```
InputBox(Prompt[, Title][, Default][, Xpos][, Ypos][, HelpFile][, Context])
```

- **Prompt** The message you want to display in the dialog box (1,024-character maximum).
- **Title** The text that appears in the dialog box title bar. The default value is the null string (nothing).
- **Default** The default value displayed in the text box. If you omit Default, the text box is displayed empty.
- **Xpos** The horizontal position of the dialog box from the left edge of the screen. The value is measured in points (there are 72 points in an inch). If you omit Xpos, the dialog box is centered horizontally.
- **Ypos** The vertical position, in points, from the top of the screen. If you omit Ypos, the dialog is centered vertically in the current window.
- **HelpFile** The text specifying the Help file that contains the custom help topic. (Again, I don’t cover Help files in this book.) If you enter HelpFile, you also have to include Context. If you include HelpFile, a Help button appears in the dialog box.
- **Context** A number that identifies the help topic in HelpFile.

For example, Listing 3.5 shows a procedure called InputBoxText that uses the InputBox method to prompt the user for data. Figure 3.4 shows the dialog box that appears. The result is stored in the inputData variable. If the user didn’t enter data, the function returns nothing, which is represented in VBA by the string value "" (this is called the null string). The procedure uses the If statement to check whether the value stored in inputData is "" and, if it’s not, it runs MsgBox to display the entered data.
Storing User Input in a Variable

CAUTION

The InputBox function returns a string (the null string) if you click Cancel. What do you do, however, if you want to use InputBox to get, say, a numeric value? This means that the result stored in your variable could be either a string or a number. The solution here is to declare your variable with the Variant type. That way, VBA will store the result—whatever it is—without generating an error.

Listing 3.5  A Procedure That Prompts the User for Input and Then Displays the Data

Sub InputBoxTest()
  Dim inputData As String

  ' Get the data
  inputData = InputBox(“Enter some text:”, “Input Box Text”)

  ' Check to see if any data was entered
  If inputData <> “” Then
    ' If so, display it
    MsgBox inputData
  End If
End Sub

Figure 3.4
A dialog box generated by the InputBox function in Listing 3.5.

From Here

- You often use operators and expressions to assign values to variables. I discuss this in detail in Chapter 4, “Building VBA Expressions.”
- Objects have a separate variable type. I talk about it, as well as about assigning objects to variables, in Chapter 5, “Working with Objects.”
- To learn about the If statement for processing MsgBox and InputBox results, see “Using If...Then to Make True/False Decisions,” p. 92.
- For MsgBox functions that use three buttons, you need to use the Select Case statement to process the result; see “Using the Select Case Statement,” p. 97.