Appendix C-1 Developing an Excel Application

An Excel application uses Excel commands, tools, and functions to perform an action. The application itself is stored as an Excel file and can only be opened from within Excel. Michael wants you to create an Excel application based on a workbook that contains stock values for a selected stock from the current year. The Report worksheet has the five macro buttons shown in Figure C-1. The first four buttons—Statistics, Line Chart, Stock Chart, and Stock History—will be used to display the corresponding worksheets in the workbook. The Statistics worksheet contains summary statistics for the price of the stock and the volume of shares traded over the past year. The Line Chart worksheet displays the closing value and volumes of shares traded over that same time interval. The Stock Chart worksheet provides an Open-High-Low-Close chart showing the stock's opening, high, low, and closing values for each day of the reporting interval. Finally, the Stock History chart provides a table showing the stock's daily values and volume of shares traded over the past year. The fifth button—Apply the Time Window—will be used to change the time interval displayed in the report. Michael wants users to choose between displaying reports for the current week, month, quarter, and year. The Statistics, Line Chart, Stock Chart, and Stock History worksheets each have a macro button—Return to Report Control—to return to the Report worksheet.

To record the Statistics macro:

1. Open the Stock workbook located in the ExcelC ▶ Tutorial folder included with your Data Files, and then save it as a macro-enabled workbook named Stock Report in the location specified by your instructor.
2. In the Documentation worksheet, enter your name and the date.

3. Display the DEVELOPER tab on the ribbon, if necessary, and then click the DEVELOPER tab.

4. Go to the **Report** worksheet.

5. In the Code group, click the **Record Macro** button. The Record Macro dialog box opens.

6. Create a macro named **Statistics** stored in **This Workbook** with **This macro displays the contents of the Statistics worksheet**. as the description, and then click the **OK** button. The dialog box closes, and you can begin to record the macro.

7. Click the **Statistics** sheet tab, and then press the **Ctrl+Home** keys to select cell A1 (even if cell A1 is already selected).

   This ensures that the macro will select cell A1 no matter what cell in the worksheet is active.

8. In the Code group, click the **Stop Recording** button. The Statistics macro is complete.

   You'll run the Statistics macro to verify that it makes the Statistics worksheet active.

9. In the Statistics worksheet, click cell **C2** to make it the active cell, and then go to the **Report** worksheet.

10. In the Code group, click the **Macros** button to open the Macro dialog box, click **Statistics** in the Macro name box if necessary, and then click the **Run** button. The Statistics worksheet becomes active and cell A1 is selected.

    Trouble? If a different worksheet is active or another cell is selected in the Statistics worksheet, you need to re-record the Statistics macro. Open the Macro dialog box, click Statistics in the Macro name box, and then click the Delete button to delete the macro. Repeat Steps 4 through 10 to record and test the macro again.
C-2 Working with the Visual Basic Editor

After you have recorded a macro, you can edit the macro code in the Visual Basic Editor. The Visual Basic Editor has three windows you can use to examine the structure and content of workbooks as well as macros. The Project Explorer displays a hierarchical list of all of the macros, worksheets, data-entry forms, and other items that make up a custom Office application. The Properties window lists the properties and values associated with each object in the Project Explorer. Finally, the Code window displays the VBA code associated with the selected item in the Project Explorer. You might see other windows, depending on how the Editor was installed on your system.

Michael wants you to review and edit the Statistics macro. You'll do this using the Visual Basic Editor.

To start the Visual Basic Editor:

1. On the DEVELOPER tab, in the Code group, click the Macros button.

2. In the Macro name box, click Statistics, if necessary, and then click the Edit button. The Visual Basic Editor opens. See Figure C-2.

Trouble? If the Project Explorer, Properties window, or the Code window is not visible, you need to display it. On the menu bar, click View, and then Project Explorer, Properties Window, or Code.
Working with Sub Procedures

VBA code is stored within procedures. There are three kinds of procedures—sub procedures, function procedures, and property procedures. A sub procedure performs an action on your project or workbook, such as formatting a cell or displaying a chart. You created a sub procedure when you recorded the Statistics macro. A function procedure returns a value and is used to create custom functions that can be used within your worksheet. A property procedure is used to create custom properties for the objects in your project. In this appendix, you'll only be working with sub procedures.

Recall that all sub procedures use the general syntax

```vba
Sub Procedure_Name(arguments)
    VBA commands and comments
End Sub
```

where `Procedure_Name` is the name of the procedure (or macro), and `arguments` pass information to the sub procedure and have roughly the same purpose as the arguments in an Excel function.

Figure C-5 shows the code the Statistics sub procedure created using the macro recorder. Although the Statistics sub procedure has no arguments, the parentheses are still required. After the name of the sub procedure, the description you entered in the Record Macro dialog box appears as a comment. A VBA comment is a statement that describes the behavior or purpose of commands in the VBA code but doesn't perform any action. VBA comments begin with an apostrophe (') and appear in green to distinguish them from other statements. After the VBA comments are the commands to select the Statistics worksheet and to select cell A1 on that worksheet. The End Sub line signals the end of the Statistics sub procedure.

Private and Public Sub Procedures

Sub procedures can be public or private. A public sub procedure is available to other modules in the project. A private sub procedure is hidden from other modules and is limited to the VBA code within its module. All sub procedures are public unless they are identified as “Private” by selecting the Private option button in the Add Procedure dialog box.
C-3a Copying and Pasting a Sub Procedure

One way to generate code quickly is to copy the code from an existing procedure. You can then edit the code, making minor changes for each new procedure. For example, the Statistics sub procedure you created displays the Statistics worksheet. Michael wants you to create additional procedures to display the other worksheets in the workbook. Because the code to display the other worksheets will be very similar to the Statistics sub procedure, you can simply copy and paste that code and then edit it for the other procedures.

**To copy and paste the Statistics sub procedure:**

1. 1.
   
   Click the title bar of the Code window to make it the active window.

2. 2.
   
   On the menu bar, click **Insert**, and then click **Procedure**. The Add Procedure dialog box opens.

3. 3.
   
   In the Name box, type **Line_Chart** as the title for the new sub procedure.

4. 4.
   
   Verify that the **Sub** and **Public** option buttons are selected to create a public sub procedure. See **Figure C-6**.

5. 5.
   
   Click the **OK** button. The beginning and ending lines of the new sub procedure are added to the Code window below a horizontal line that separates the new procedure from the Statistics sub procedure.

You add, delete, and replace text in the Code window the same way you do in any text editor. You could type the entire code for the new macro. But because you already recorded the Statistics macro, you can copy and paste the code of the Statistics procedure.
To copy the VBA code from the Statistics sub procedure:

1. 1.
   Select all the comment lines and VBA commands from the Statistics sub procedure, beginning with the first apostrophe directly below the Sub Statistics() line and ending with the Range(“A1”). Select line. Do not select either the Sub Statistics() or the End Sub line.

2. 2.
   On the Standard toolbar, click the **Copy** button (or press the **Ctrl+C** keys). The selected code is copied to the Clipboard.

3. 3.
   In the Code window, click the blank line below the Public Sub Line_Chart() command line.

4. 4.
   On the Standard toolbar, click the **Paste** button (or press the **Ctrl+V** keys). The Statistics sub procedure code is pasted into the Line_Chart sub procedure.

You need to edit the pasted code. You could type individual changes in the code or you can use the Replace dialog box to replace multiple instances of text in the code. You can choose to replace the text throughout the current procedure, the current module (across several procedures), or the current project (across several modules) by clicking the corresponding option button in the Replace dialog box.

Michael wants you to edit the code to create a macro that selects cell A1 in the Line Chart worksheet. You'll replace all occurrences of “Statistics” with “Line Chart.”

To replace text in the Line_Chart sub procedure:

1. 1.
   Click in the Line_Chart sub procedure, if necessary.

2. 2.
   On the menu bar, click **Edit**, and then click **Replace** (or press the **Ctrl+H** keys). The Replace dialog box opens.

3. 3.
   In the Find What box, type **Statistics**, and then press the **Tab** key.

4. 4.
   In the Replace With box, type **Line Chart**.

5. 5.
In the Search section, click the **Current Procedure** option button to specify that text is replaced only within the current procedure—in this case, the Line_Chart procedure. See Figure C-7.

Figure C-7 Replace dialog box to edit the Line_Chart sub procedure

6. Click the **Replace All** button. A dialog box indicates that three replacements of the Statistics text were made.

7. Click the **OK** button, and then click the **Cancel** button to close the Replace dialog box. The Line_Chart sub procedure is edited.

C-3b Running a Sub Procedure

After creating a procedure, you should test it to ensure that it works as intended. You can test a macro by running it from the workbook or from the Visual Basic Editor. Michael wants you to run the Line_Chart sub procedure to verify that it opens the Line Chart worksheet. You'll do this from the Visual Basic Editor.

**To run the Line_Chart sub procedure in the Visual Basic Editor:**

1. Make sure the insertion point is still within the Line_Chart sub procedure.

2. On the menu bar, click **Run**, and then click **Run Sub/UserForm** (or press the F5 key). The Visual Basic Editor runs the sub procedure selected in the Code window—in this case, the Line_Chart sub procedure.

Trouble? If the Visual Basic Editor displays an error message, you probably made a mistake while creating the Line_Chart sub procedure. Click the End button in the dialog box, make sure Line Chart is spelled correctly in all three instances, correct any errors you find, and then repeat Steps 1 and 2.

3. Return to Microsoft Excel, and verify that the Line Chart worksheet is now the active sheet in the workbook.

4. 


Save the workbook, and then press the Alt+F11 keys to return to the Visual Basic Editor.

You'll use the same process to create macros that display the remaining worksheets in the Tornado Analysis workbook. You'll copy and edit the sub procedure to display the contents of the Stock Chart, Stock History, and Report Control worksheets.

**To create the remaining sub procedures:**

1. 1. Make sure the Code window in the Visual Basic Editor is active.

2. 2. On the menu bar, click **Insert**, and then click **Procedure**. The Add Procedure dialog box opens.

3. 3. Type **Stock_Chart** in the Name box, verify that the **Sub** and **Public** option buttons are selected, and then click the **OK** button.

4. 4. On the Standard toolbar, click the **Paste** button (or press the Ctrl+V keys). The lines of code you copied from the Statistics sub procedure are pasted into the new sub procedure.

5. 5. On the menu bar Click **Edit**, and then click **Replace** (or press the Ctrl+H keys). The Replace dialog box opens.

6. 6. Click the **Find What** arrow, click **Statistics**, type **Stock Chart** in the Replace With box, verify that the **Current Procedure** option button is selected, and then click the **Replace All** button.

7. 7. Click the **OK** button to confirm that three replacements were made, and then click the **Cancel** button to close the Replace dialog box.

8. 8. Repeat Steps 2 through 7 to create a sub procedure named **Stock_History**, replacing Statistics with **Stock History** in the pasted program code.

9. 9. Repeat Steps 2 through 7 to create a sub procedure named **Report_Control**, replacing Statistics with **Report Control** in the pasted program code. **Figure C-8** show the code for the three sub procedures you just created. You'll need to scroll the Code window to see all of the code.
C-3c Assigning a Sub Procedure to a Macro Button

You have created five sub procedures to display each worksheet in the Stock Report workbook. You'll return to Excel and assign those sub procedures to macro buttons.

To assign the sub procedures to macro buttons:

1. Return to Excel, and go to the **Report Control** worksheet.

2. Right-click the **Statistics** macro button and then click **Assign Macro**. The Assign Macro dialog box opens.

3. In the Macro name box, click **Statistics** and then click the **OK** button. The Statistics macro is assigned to the Statistics macro button.

4. Press the **Esc** key, and then click the **Statistics** button. The Statistics worksheet becomes the active sheet in the workbook with cell A1 selected.
Trouble? If the worksheet doesn't open, the macro button might be selected (indicated by selection handles around the button). If the button is selected, press the Esc key, and then click the button again.

5. Right-click the Return to Report Control macro button, click Assign Macro to open the Assign Macro dialog box, select Report Control as the macro name, and then click the OK button. The Report Control macro is assigned to the button.

6. Press the Esc key, and then click the Return to Report Control button. The Report Control worksheet becomes the active sheet in the workbook.

7. Repeat Steps 2 through 6 to assign the appropriate macros to the Line Chart, Stock Chart, and Stock History buttons on the Report Control worksheet to open the Line Chart, Stock Chart and Stock History worksheets.

8. On the Line Chart, Stock Chart and Stock History worksheets, repeat Steps 2 through 6 to assign a macro to the Return to Report Control button to return the user to the Report Control worksheet.

9. Save the workbook.

Now that you have created a few sub procedures using the Visual Basic Editor, you'll examine the VBA commands in those procedures.

Proskills

**Problem Solving: Learning VBA**

The best way to learn VBA is by doing. Start with the macro recorder to generate VBA code. The code is guaranteed to be free of syntax errors, so you can identify the key elements of the VBA language. However, the macro recorder does not write the most efficient code. It also records commands and actions that you might not want or need in a finished project. Experienced programmers might use the macro recorder as a starting point, but then edit the generated code to remove the extraneous material.

Another good source of information is the Microsoft Visual Basic for Applications Help. Help contains descriptions of all of the VBA commands and operations. It also includes code samples you can often apply to your own projects.

Next, try writing some code. The interactive tools guide you to use the correct syntax by identifying syntax errors as you type them.

Finally, examine what other programmers are doing. In many cases, program code is password-protected. You can learn from others' techniques, and the web is a great source of VBA code samples. However, if you do use
code from another programmer, be sure to obtain permission and to cite the programmer's work in any publication.

C-4 An Introduction to Visual Basic for Applications

VBA is an object-oriented programming language, in which tasks are performed by manipulating objects. Almost anything in Excel—from a single cell, to an entire worksheet, to the Excel application itself—is considered a Visual Basic object. You can perform any task on these objects that you can perform in Excel, such as creating charts, moving worksheets, or entering formulas into cells. Figure C-9 describes some of the common Excel objects used in many VBA programs.

Figure C-9

VBA objects in Excel

<table>
<thead>
<tr>
<th>Excel Object</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>A range in a worksheet</td>
</tr>
<tr>
<td>Name</td>
<td>A defined name in a workbook</td>
</tr>
<tr>
<td>Chart</td>
<td>A chart in a workbook (either embedded within a worksheet or stored as a chart sheet)</td>
</tr>
<tr>
<td>ChartObject</td>
<td>A chart embedded within a worksheet</td>
</tr>
<tr>
<td>Worksheet</td>
<td>A worksheet in a workbook</td>
</tr>
<tr>
<td>Workbook</td>
<td>An Excel workbook</td>
</tr>
<tr>
<td>VBAProject</td>
<td>A VBA project</td>
</tr>
<tr>
<td>Application</td>
<td>The Excel program itself</td>
</tr>
</tbody>
</table>

Objects are often grouped into collection objects, which are themselves objects. For example, a sheet is an object, but the collection of all the sheets in a workbook is also an object. To refer to a specific object in a collection, use the object reference

\[
\text{object\_collection(id)}
\]

where \text{object\_collection} is the name of the object collection and \text{id} is either a name or number that identifies an object in the collection. For example, the object collection Sheets refers to all of the sheets in a particular workbook. The reference to the Statistics worksheet out of that collection is

\[
\text{Sheets("Statistics")}
\]

You could also use the object reference Sheets(6) because the Statistics worksheet is the sixth object in the collection of worksheets. Note that the number does not indicate the sheet's location in the workbook. For example, the Statistics worksheet is the sixth worksheet in the collection, but it might be moved to any location in the workbook. Figure C-10 provides other examples of references to objects within collections.

Figure C-10

Object collection examples
VBA organizes objects and object collections in a hierarchy with the Excel application at the top and the individual cells of a workbook at the bottom. This hierarchy is often referred to as the Excel Object Model. The general syntax for referencing an object's location within this hierarchy is

\[ \text{object1.object2.object3} \]

where \( \text{object1} \) is an object at the upper level of the hierarchy, \( \text{object2} \) is the object at the next lower level in the hierarchy, \( \text{object3} \) is the next lower-level object, and so forth. For example, the complete object reference to cell A1 in the Info worksheet of the Stocks workbook within the Excel application is:

\[ \text{Application.Workbooks("Stocks").Sheets("Info").Range("A1")} \]

If you don't include the complete object hierarchy, the object is assumed to be based in the active application, workbook, and worksheet. For example, the following object reference refers to cell A1 of the Info worksheet, and the workbook is assumed to be the active workbook within the Excel application:

\[ \text{Sheets("Info").Range("A1")} \]

Similarly, the following object reference refers to cell A1 of the active sheet in the active workbook:

\[ \text{Range("A1")} \]

To help you work with the hierarchy of objects in the Excel Object Model, VBA provides special object names to refer directly to certain objects. Figure C-11 describes some of these special object names.

Figure C-11

Special object names in VBA

<table>
<thead>
<tr>
<th>Object Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveCell</td>
<td>The currently selected cell</td>
</tr>
<tr>
<td>ActiveChart</td>
<td>The currently selected chart</td>
</tr>
<tr>
<td>ActiveSheet</td>
<td>The currently selected sheet</td>
</tr>
<tr>
<td>ActiveWindow</td>
<td>The currently selected window</td>
</tr>
<tr>
<td>ActiveWorkbook</td>
<td>The current workbook</td>
</tr>
<tr>
<td>ThisCell</td>
<td>The cell from which a custom function is being run</td>
</tr>
</tbody>
</table>
In VBA, the following two references are equivalent because they reference cell A1 of the active sheet and workbook:

```
Range("A1")
ActiveWorkbook.ActiveSheet.Range("A1")
```

The ActiveWorkbook refers to the workbook in which the macro is running; the ThisWorkbook object refers to the workbook in which the macro code has been stored.

## C-4a Modifying Properties

The VBA language alters objects by modifying the object's properties. For example, a worksheet cell supports several properties, such as the value or formula contained in the cell, the formatting applied to the cell's appearance, and the text of the comment that might be attached to the cell. Figure C-12 describes some Excel objects and the properties associated with them. Note that some properties are themselves objects.

### Figure C-12

#### Objects and their properties

<table>
<thead>
<tr>
<th>Object</th>
<th>Property</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>Address</td>
<td>The cell reference of the range</td>
</tr>
<tr>
<td></td>
<td>Comment</td>
<td>A comment attached to the cell</td>
</tr>
<tr>
<td></td>
<td>Formula</td>
<td>The formula entered into the cell</td>
</tr>
<tr>
<td></td>
<td>Value</td>
<td>The value of the cell</td>
</tr>
<tr>
<td></td>
<td>RefersTo</td>
<td>The cell(s) that the defined name refers to</td>
</tr>
<tr>
<td>Name</td>
<td>Value</td>
<td>The value of the cell referred to by the defined name</td>
</tr>
<tr>
<td>Worksheet</td>
<td>Name</td>
<td>The name of the worksheet</td>
</tr>
<tr>
<td></td>
<td>Visible</td>
<td>Whether the worksheet is visible or hidden</td>
</tr>
<tr>
<td></td>
<td>ChartTitle</td>
<td>The text of the chart's title</td>
</tr>
<tr>
<td>Chart</td>
<td>ChartType</td>
<td>The type of the chart</td>
</tr>
<tr>
<td></td>
<td>HasLegend</td>
<td>Whether the chart has a legend</td>
</tr>
<tr>
<td></td>
<td>HasPassword</td>
<td>Whether the workbook has a password</td>
</tr>
<tr>
<td>Workbook</td>
<td>Name</td>
<td>The name of the workbook</td>
</tr>
<tr>
<td></td>
<td>Path</td>
<td>The folder and drive in which the workbook is stored</td>
</tr>
<tr>
<td></td>
<td>Saved</td>
<td>Whether the workbook has been saved</td>
</tr>
</tbody>
</table>

This is only a small sample of the objects and properties available in VBA programs. Literally everything in Excel can be expressed in terms of an object or a property.

- Object properties can be changed using the VBA statement
object.property = expression

where object is a reference to an object, property is the name of the object's property, and expression is a value that you want to assign to the property. For example, the following command changes the value of cell A2 in the active sheet to 395:

Range("A2").Value = 395

Figure C-13 shows other VBA statements that change the property value of Excel objects.

Figure C-13

Examples of changing a property's value

<table>
<thead>
<tr>
<th>VBA Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>ActiveCell.Value = 23</td>
<td>Changes the value of the active cell to 23</td>
</tr>
<tr>
<td>Sheets(&quot;Table&quot;).Range(&quot;A3&quot;).Value = Sheets(&quot;History&quot;).Range(&quot;B3&quot;).Value</td>
<td>Set the value of cell A3 in the Table worksheet equal to the value of cell B3 in the History worksheet</td>
</tr>
<tr>
<td>Range(&quot;A5&quot;).Font.Italic = true</td>
<td>Displays the text of cell A5 in an italic font</td>
</tr>
<tr>
<td>Worksheets(&quot;Raw Data&quot;).Name = &quot;Table&quot;</td>
<td>Changes the name of the Raw Data worksheet to Table</td>
</tr>
<tr>
<td>ActiveWorkbook.Password = &quot;stocks&quot;</td>
<td>Changes the password of the current workbook to stocks</td>
</tr>
<tr>
<td>Application.StatusBar = &quot;Running macro&quot;</td>
<td>Changes the status bar text to &quot;Running macro&quot;</td>
</tr>
<tr>
<td>Application.StatusBar = false</td>
<td>Resets the status bar text to its default value</td>
</tr>
<tr>
<td>Application.ScreenUpdating = false</td>
<td>Turns off screen updating within Excel</td>
</tr>
<tr>
<td>Application.ScreenUpdating = true</td>
<td>Turns on screen updating within Excel</td>
</tr>
</tbody>
</table>

© 2014 Cengage Learning

You can also use an object property statement to turn a property on or off. The following VBA command hides the Documentation worksheet from the user by setting the sheet's Visible property to false; to make the worksheet visible again, you switch the value to true:

Sheets("Documentation").Visible = false

C-4b Applying Methods

A method is an action that can be performed on an object, such as closing a workbook or printing the contents of a worksheet. The syntax to apply a method is

object.method
where object references an Excel object and method is the name of the VBA method that can be applied to the object. For example, the following commands from the Stock_Chart sub procedure use the Select method to select the Stock Chart worksheet from the Sheets collection and then select cell A1 in that worksheet:

```vba
Sheets("Statistics").Select
Range("A1").Select
```

**Figure C-14** describes some of the other methods associated with different VBA objects.

**Objects and their methods**

<table>
<thead>
<tr>
<th>Object</th>
<th>Method</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range</td>
<td>ClearContents</td>
<td>Clears all formulas and values in the range</td>
</tr>
<tr>
<td></td>
<td>Copy</td>
<td>Copies the values in the range to the Clipboard</td>
</tr>
<tr>
<td></td>
<td>Merge</td>
<td>Merges the cells in the range</td>
</tr>
<tr>
<td>Worksheet</td>
<td>Delete</td>
<td>Deletes the worksheet</td>
</tr>
<tr>
<td></td>
<td>Select</td>
<td>Selects (and displays) the worksheet</td>
</tr>
<tr>
<td></td>
<td>Close</td>
<td>Closes the workbook</td>
</tr>
<tr>
<td>Workbook</td>
<td>Protect</td>
<td>Protects the workbook</td>
</tr>
<tr>
<td></td>
<td>Save</td>
<td>Saves the workbook</td>
</tr>
<tr>
<td></td>
<td>Copy</td>
<td>Copies the chart to the Clipboard</td>
</tr>
<tr>
<td>Chart</td>
<td>Select</td>
<td>Selects the chart</td>
</tr>
<tr>
<td></td>
<td>Delete</td>
<td>Deletes the chart</td>
</tr>
<tr>
<td>Charts</td>
<td>Select</td>
<td>Selects the chart sheets in the workbook</td>
</tr>
<tr>
<td>Worksheets</td>
<td>Select</td>
<td>Selects the worksheets in the workbook</td>
</tr>
</tbody>
</table>

© 2014 Cengage Learning

Methods often have parameters that govern how they are applied. For example, the workbook object has the SaveAs method for saving the workbook to a file. But to apply the SaveAs method, you need to supply a filename. The syntax to apply parameter values to a method is

```vba
object.method parameter1:=value1 parameter2:=value2...
```

where `parameter1` and `parameter2` are the names of parameters associated with `method`, and `value1` and `value2` are the values assigned to those parameters. So, the following VBA command saves the active workbook using the filename Budget.xlsx:

```vba
ActiveWorkbook.SaveAs Filename:="Budget.xlsx"
```

**Figure C-15** describes other ways of applying methods with parameter values to an object.

**Code to apply a method to an Excel object**
<table>
<thead>
<tr>
<th>VBA Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Range(&quot;Al&quot;).Copy</td>
<td>Copies the contents of cell A1 into cell A5</td>
</tr>
<tr>
<td>Destination:=Range(&quot;A5&quot;)</td>
<td></td>
</tr>
<tr>
<td>Sheets(&quot;Orders&quot;).Range(&quot;C5:C10, E10:E11&quot;).ClearContents</td>
<td>Clear the contents of cells C5:C10 and E10:E11 in the Orders worksheet</td>
</tr>
<tr>
<td>Sheets(&quot;Sheet1&quot;).Move</td>
<td>Moves the Sheet1 worksheet after the Sheet3 worksheet</td>
</tr>
<tr>
<td>After:=Sheets(&quot;Sheet3&quot;)</td>
<td></td>
</tr>
<tr>
<td>ActiveWorkbook.SaveAs</td>
<td>Saves the active workbook with the filename “Stocks”</td>
</tr>
<tr>
<td>Filename:=&quot;Stocks&quot;</td>
<td></td>
</tr>
<tr>
<td>ActiveSheet.Protect Password:=&quot;penguin&quot;</td>
<td>Protects the active worksheet using the password “penguin”</td>
</tr>
<tr>
<td>ActiveSheet.Unprotect</td>
<td>Unprotects the active sheet using the password “penguin”</td>
</tr>
<tr>
<td>Password:=&quot;penguin&quot;</td>
<td></td>
</tr>
<tr>
<td>Workbooks.Open Filename:=&quot;Budget.xlsx&quot;</td>
<td>Opens the Budget.xlsx file, adding it to the collection of open workbooks</td>
</tr>
</tbody>
</table>

© 2014 Cengage Learning

Another way of entering an object method is to use the following syntax, which requires that the parameter values be entered in a specific order as determined by the syntax for the method that is being applied:

```
object.method(value1, value2, ...)
```

However, you must know exactly what parameters are required for the method and in what order they need to be entered. Thus, to save the active workbook in the Budget.xlsx file, you could run the following VBA command:

```
ActiveWorkbook.SaveAs("Budget.xlsx")
```

Which syntax you use is often a matter of personal preference.

### C-5 Writing a Visual Basic Sub Procedure

Once you've learned the basic concepts of the VBA programming language, it is often more efficient to write your own sub procedures rather than using the macro recorder.

Michael wants you to write a sub procedure that displays results for different time windows in the stock report application. To create this program, you first must understand how Michael organized the data in the Stock Report workbook. Michael defined range names for the data columns in the Stock History worksheet. Figure C-16 lists the defined names that track the stock values throughout the current year.

Figure C-16

**Defined range names for stock values over the year**
<table>
<thead>
<tr>
<th>Defined Name</th>
<th>Definition</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>YearDates</td>
<td>=$B$5:$B$255</td>
<td>Dates on which stock values were recorded throughout the year</td>
</tr>
<tr>
<td>YearOpen</td>
<td>=$C$5:$C$255</td>
<td>Daily opening values</td>
</tr>
<tr>
<td>YearHigh</td>
<td>=$D$5:$D$255</td>
<td>Daily high values</td>
</tr>
<tr>
<td>YearLow</td>
<td>=$E$5:$E$255</td>
<td>Daily low values</td>
</tr>
<tr>
<td>YearClose</td>
<td>=$F$5:$F$255</td>
<td>Daily closing values</td>
</tr>
<tr>
<td>YearVolume</td>
<td>=$G$5:$G$255</td>
<td>Daily volume of shares traded</td>
</tr>
</tbody>
</table>

© 2014 Cengage Learning

Michael also defined names for the current week, month, and quarter. For example, the stock dates for the current week are referenced with the defined name WeekDates while the opening values from the current week are referenced with the defined name WeekOpen. The month values are referenced with MonthDates and MonthOpen, while the quarter values are referenced with QuarterDates and QuarterOpen. The other columns are similarly named.

To create the charts and table of statistics, Michael uses the defined names listed in Figure C-17. For example, the formula to calculate the highest stock value for the current time period is:

\[ \text{MAX(StockHigh)} \]

Because the defined name StockHigh currently has the value =YearHigh, this formula returns the highest stock value for the current year. To change the time period displayed in the charts and statistics, Michael simply changes the definition of the defined names in Figure C-17. For example, to create charts and statistics for the current week, the definition of the StockDates range name changes from =YearDates to =WeekDates, the definition of StockOpen changes from =YearOpen to =WeekOpen, and so forth. All of the charts and statistics will automatically reflect the new time period under the revised values of the Stock range names.

Figure C-17

**Defined names for stock statistics**

<table>
<thead>
<tr>
<th>Defined Name</th>
<th>Definition</th>
<th>Refers to</th>
</tr>
</thead>
<tbody>
<tr>
<td>StockDates</td>
<td>=YearDates</td>
<td>Dates displayed in the stock report's charts and statistics</td>
</tr>
<tr>
<td>StockOpen</td>
<td>=YearOpen</td>
<td>Daily opening values</td>
</tr>
<tr>
<td>StockHigh</td>
<td>=YearHigh</td>
<td>Daily high values</td>
</tr>
<tr>
<td>StockLow</td>
<td>=YearLow</td>
<td>Daily low values</td>
</tr>
<tr>
<td>StockClose</td>
<td>=YearClose</td>
<td>Daily closing values</td>
</tr>
<tr>
<td>StockVolume</td>
<td>=YearVolume</td>
<td>Daily volume of shares traded</td>
</tr>
</tbody>
</table>

© 2014 Cengage Learning
To see this in action, you could use the Name Manager dialog box in Excel. Instead, you'll write a sub procedure to change the time window from its current one-year window to one week. This sub procedure will use the Names object collection along with the Value property to change the value of each range name. So, to change the value of the StockDates range name to =WeekDates and StockOpen to =WeekOpen, you would run the following commands:

```
Names("StockDates").Value="=WeekDates"
Names("StockOpen").Value="=WeekOpen"
```

Commands to change the values of the StockHigh, StockLow, StockClose, and StockVolume range names are similar.

The stock report application also identifies the size of the time window in cell C5 of the Report Control worksheet. Currently, that cell has the value YEAR because the current workbook is showing stock information for the entire year. To change the value of this cell to WEEK, you would use the following object hierarchy to point to cell C5 in the Report Control worksheet, changing its value using the Value property

```
Sheets("Report Control").Range("C5").Value = "WEEK"
```

Michael wants you to use these commands to write the new sub procedure.

**To write a sub procedure to change the range name values:**

1. Press the Alt+F11 keys to return to the Visual Basic Editor.

2. On the menu bar, click Insert, and then click Procedure. The Add Procedure dialog box opens.

3. Type Change_Time in the Name box, verify that the Sub and Public option buttons are selected, and click the OK button. The Change_Time sub procedure is added to the Code window.

4. Type the following lines of code, pressing the Tab key at the beginning of each line to make the code easier to read:

```
Sheets("Report Control").Range("C5").Value = "WEEK"
Names("StockDates").Value = "=WeekDates"
Names("StockOpen").Value = "=WeekOpen"
Names("StockHigh").Value = "=WeekHigh"
Names("StockLow").Value = "=WeekLow"
Names("StockClose").Value = "=WeekClose"
Names("StockVolume").Value = "=WeekVolume"
```

Figure C-18 shows the code of the Change_Time sub procedure.
Next, you'll test your code to verify that the charts and statistics in the Stock Report workbook change to reflect the new time window.

**To test the Change_Time sub procedure:**

1. Verify that the insertion point is still within the Change_Time sub procedure, and then press the F5 key to run the code.

   Trouble? If an error message appears, check your code against the code shown in Figure C-18. Common errors include not enclosing values within quotation marks or not adding the = symbol within the defined name value.

2. Return to Excel, and then go to the Report Control worksheet. Notice that the word WEEK appears in cell C5.

3. Click the Line Chart macro button to go to the Line Chart worksheet. The chart displays the closing values and volume for the week. See Figure C-19.

Figure C-19Line Chart worksheet showing values from the current week
4. Go to the other sheets in the workbook and confirm that the Statistics and Stock Chart worksheets summarize the values for the current week.

The Change_Time sub procedure works effectively to show the stock values for the current week, but Michael wants a more general macro that will work with other time intervals. You can create such a macro using variables.

Insight

Using the Immediate Window

As you work in VBA, you might want to see the effects of a single command rather than an entire sub procedure. You can run single command lines in the Visual Basic Editor Immediate window. To open the Immediate window, click View on the VBA menu bar, and then click Immediate Window (or press the Ctrl+G keys). The Immediate window opens and you can then type your VBA commands directly into this window. Every time you reach the end of a command and press the Enter key, the command runs and you obtain immediate feedback on the effects of the command.

C-6a Declaring a Variable

Variables are declared using the command

```
Dim variable
```

where `variable` is the variable name. The Dim in this command is short for “dimensioning,” which refers to the process of allocating memory space for the variable. Variables can store a wide variety of data from text to numbers to objects. You specify the type of data stored in the variable using the command

```
Dim variable as type
```
where *type* is the data type. For example, a variable containing text is declared using the statement

```vba
Dim SheetName as String
```

where String is the data type for text values. Other data types include Date for date values, Integer for integers, Decimal for whole or decimal numbers, and Object for Excel objects such as workbooks, worksheets, or cells. You can learn more about data types in Excel Help.

**Insight**

### Determining a Data Type

It is not strictly required to specify a data type or declare a variable in Visual Basic. If you don't specify a data type, Excel determines the data type by what is stored in the variable. Likewise, if you don't declare a variable, Excel creates the variable when you first attempt to store data in it. However, good programming practice is to declare variables and their data types as a way of catching errors that might creep into the code.

**C-6b Assigning a Value to a Variable**

After you declare a variable, you store data in it using the command

```vba
Variable = expression
```

where *expression* is the initial value or text that is assigned to the variable. For example, the following command stores the text “Budget” in the variable SheetName. If you have not declared the SheetName variable beforehand, this command also declares the variable:

```vba
SheetName = "Budget"
```

Because variables can also store objects such as worksheets, workbooks, or ranges, the command

```vba
Set variable = object
```

stores an object in a variable, where *object* is an Excel object. For example, the following command stores a reference to the Statistics worksheet object in the StatsSheet variable.

```vba
Set StatsSheet = Sheets("Statistics")
```

One advantage of using object variables is that you can take a long extended object reference and reduce it to a compact variable name. **Figure C-20** shows VBA statements in which variables are assigned values or are used to store objects.

**Figure C-20**

**Code to set the value of a variable**

<table>
<thead>
<tr>
<th>VBA Code</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Year = 2016</td>
<td>Stores the value 2016 in the Year variable</td>
</tr>
<tr>
<td>time_window = “MONTH”</td>
<td>Stores the text string “MONTH” in the time_window variable</td>
</tr>
<tr>
<td>VBA Code</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>time_window = Range(“C5”).Value</code></td>
<td>Stores the value entered in cell C5 of the active worksheet in the <code>time_window</code> variable</td>
</tr>
<tr>
<td><code>Set WSheet = Sheets(“Statistics”)</code></td>
<td>References the Statistics worksheet using the <code>WSheet</code> variable</td>
</tr>
<tr>
<td><code>Set WBook = Workbooks(“Stock Report”)</code></td>
<td>References the Stock Report workbook using the <code>WBook</code> variable</td>
</tr>
</tbody>
</table>

Michael wants you to create a variable named `time_window` for the Change_Time sub procedure that uses whatever text is entered into cell C5 of the Report worksheet as the value of this variable. You'll declare the `time_window` variable and set its initial value now.

**To create the `time_window` variable:**

1. 1. Press the **Alt+F11** keys to return to the Visual Basic Editor.

2. 2. In the Code window, click the end of the Public Sub Change_Time() line, and then press the **Enter** key to insert a new blank line.

3. 3. Press the **Tab** key to indent the line, type **Dim time_window**, and then press the **Enter** key.

4. 4. Type `time_window = Sheets(“Report Control”).Range(“C5”).Value` and then press the **Enter** key. See Figure C-21.

Figure C-21 Code window with the `time_window` variable

Next, you will use the variable that you declared as part of the Change_Time sub procedure.
Problem Solving: Common Sources of Program Errors

VBA programs can lead to three types of errors:

- **Syntax errors** occur when a line of VBA code is entered improperly. The Visual Basic Editor catches such errors before you get a chance to run the program and usually provides ScreenTips to help you fix the error.
- **Run-time errors** occur when the program is running. At that point, the code has no known syntax errors, but something in the code keeps it from running successfully. For example, Excel does not recognize a mistyped worksheet name as a programming error, but the code will fail when it cannot locate the sheet object the program references. You can use the Visual Basic Editor to run the program up to a certain spot in the code. By running only part of the code, you can quickly locate the command line that is causing the program to fail. After you identify the command, you can use Excel Help to determine the reason for the error.
- **Logical errors** can be the most difficult to resolve. With these errors, the program is free of syntax errors and it runs without failing, but the results are incorrect. To fix a logical error, run the program one line at a time, confirming the accuracy of each operation until you find the command that results in an incorrect value or operation. If you still cannot find the source of the error, consider a different approach to solving the problem. VBA often provides several different ways of performing the same task.

C-6c Referencing a Variable

Once a variable has been declared and assigned a value you can use it with any for VBA command. For example, if you set the TitleText variable to reference cell A1 in the Statistics worksheet

Set TitleText = Sheets("Statistics").Range("A1")

both of the following commands will change the value stored in that cell to “Statistics Report”:

TitleText.Value = "Statistics Report"

In the following example, the initial_number variable is set to 100. When this is referenced in the second command, 150 is stored in the next_number variable:

initial_number = 100
next_number = initial_number + 50

Finally, you can combine variables that contain text using the ampersand (&) character. The following VBA commands combine the text “Coolidge”, a space, and the text “Financial.” The result is that the Title_Line variable contains the text “Coolidge Financial.”

Word1 = "Coolidge"
Word2 = "Financial"
Title_Line = Word1 & " " & Word2

Michael wants to use the ampersand character along with the time_window variable to set the value of the different Stock names in the workbook. For example, instead of writing the command
you'll use:

```
Names("StockDates").Value = ":WeekDates"
```

where the `time_window` variable will contain the text string “WEEK”, “MONTH”, “QUARTER” or “YEAR”. You'll revise the Change_Time sub procedure to use the `time_window` variable to set the values of the different Stock range names in the workbook.

To reference the `time_window` variable:

1. Select the line `Sheets("Report Control").Range("C5").Value = "WEEK"` and then press the Delete key. You no longer need this line in the function.

2. On the menu bar, click Edit, and then click Replace (or press the Ctrl+H keys). The Replace dialog box opens.

3. In the Find What box, type “=Week and then press the Tab key.

4. In the Replace With box, type “=" & `time_window` & ".

5. Make sure that the Current Procedure option button is selected. See Figure C-22.

Figure C-22Replace dialog box

6. Click the Replace All button. All occurrences of “=Week are replaced in the Change_Time sub procedure.

7. Click the Replace All button. All occurrences of “=Week are replaced in the Change_Time sub procedure.
Click the OK button to confirm that six replacements were made, and then click the Cancel button to return to the Code window. See Figure C-23.

Figure C-23 Final Change_Time sub procedure

The Change_Time sub procedure is complete. Michael wants you to assign the sub procedure to a macro button and then test it to ensure that you can display weekly, monthly, quarterly, and yearly stock reports.

To assign the Change_Time macro and test it:

1. On the menu bar, click File, and then click Close and Return to Microsoft Excel (or press the Alt+Q keys).

2. Go to the Report Control worksheet, right-click the Apply the Time Window macro button, and then click Assign Macro on the shortcut menu.

3. In the Macro name box, click Change_Time, and then click the OK button.

4. Press the Esc key to deselect the macro button.

5. In cell C5, select MONTH from the list of possible values.

6. Click the Apply the Time Window macro button. The worksheets in the Statistics, Line Chart, and Stock Chart worksheets now show the month results.
7. Click the **Line Chart** macro button to go to the Line Chart worksheet. The chart shows the closing values and volume of the current month. See Figure C-24.

Figure C-24 Line Chart worksheet for the current month

8. View the **Stock Chart** and **Statistics** worksheets, and verify that they summarize the stock values for the current month.

Next you'll switch the time window to the current quarter.

9. Go to the **Report Control** worksheet, select **QUARTER** in cell C5, and then click the **Apply the Time Window** button to generate the quarterly reports.

10. Go to the **Statistics, Line Chart**, and **Stock Chart** worksheets and verify that their content changed to show the quarterly data.

11. Save the workbook, and then close it.

The Stock Report workbook is complete. Using the Visual Basic Editor, you created macros so that users can easily switch between worksheets and change the time window.
Sharing Macros between Workbooks

If you need to use the same macro for different workbooks, you can save time by sharing the macro in a special workbook containing your personal macros. The personal macro workbook has the filename Personal.xlsb and is stored in the C:\Users\username\AppData\Roaming\Microsoft\Excel\XLSTART folder where username is your user name. This is a hidden folder. To access it directly, you must set Windows to display hidden files and folders. You do not need to do so because Excel will automatically open the Personal.xlsb file each time you start Excel, making all the macros stored in that file available to any open workbooks.

To record a personal macro and create the Personal.xlsb file if it is not already created, use the macro recorder and choose Personal Macro Workbook from the Store macro list box in the Record Macro dialog box, and then record the macro steps. After you have created one macro in the Personal.xlsb file, you can use the Visual Basic Editor to revise the macro and create other sub procedures. You can also copy the Personal.xlsb file from one computer to another to share your macros with other users.

C-6c Referencing a Variable

Once a variable has been declared and assigned a value you can use it with any for VBA command. For example, if you set the TitleText variable to reference cell A1 in the Statistics worksheet

```vba
Set TitleText = Sheets("Statistics").Range("A1")
```

both of the following commands will change the value stored in that cell to “Statistics Report”:

```vba
TitleText.Value = "Statistics Report"
```

In the following example, the initial_number variable is set to 100. When this is referenced in the second command, 150 is stored in the next_number variable:

```vba
initial_number = 100
next_number = initial_number + 50
```

Finally, you can combine variables that contain text using the ampersand (&) character. The following VBA commands combine the text “Coolidge”, a space, and the text “Financial.” The result is that the Title_Line variable contains the text “Coolidge Financial.”

```vba
Word1 = "Coolidge"
Word2 = "Financial"
Title_Line = Word1 & " " & Word2
```

Michael wants to use the ampersand character along with the time_window variable to set the value of the different Stock names in the workbook. For example, instead of writing the command

```vba
Names("StockDates").Value = "=WeekDates"
```

you'll use:

```vba
Names("StockDates").Value = "=" & time_window & "Dates"
```
where the time_window variable will contain the text string “WEEK”, “MONTH”, “QUARTER” or “YEAR”. You’ll revise the Change_Time sub procedure to use the time_window variable to set the values of the different Stock range names in the workbook.

To reference the time_window variable:

1. 1.

   Select the line `Sheets("Report Control").Range("C5").Value = "WEEK"` and then press the Delete key. You no longer need this line in the function.

2. 2.

   On the menu bar, click Edit, and then click Replace (or press the Ctrl+H keys). The Replace dialog box opens.

3. 3.

   In the Find What box, type “=Week” and then press the Tab key.

4. 4.

   In the Replace With box, type “=” & time_window & “.”

5. 5.

   Make sure that the Current Procedure option button is selected. See Figure C-22.

Figure C-22Replace dialog box

6. 6.

   Click the Replace All button. All occurrences of “=Week” are replaced in the Change_Time sub procedure.

7. 7.

   Click the OK button to confirm that six replacements were made, and then click the Cancel button to return to the Code window. See Figure C-23.

Figure C-23Final Change_Time sub procedure
The Change_Time sub procedure is complete. Michael wants you to assign the sub procedure to a macro button and then test it to ensure that you can display weekly, monthly, quarterly, and yearly stock reports.

**To assign the Change_Time macro and test it:**

1. On the menu bar, click **File**, and then click **Close and Return to Microsoft Excel** (or press the **Alt+Q** keys).

2. Go to the **Report Control** worksheet, right-click the **Apply the Time Window** macro button, and then click **Assign Macro** on the shortcut menu.

3. In the Macro name box, click **Change_Time**, and then click the **OK** button.

4. Press the **Esc** key to deselect the macro button.

5. In cell **C5**, select **MONTH** from the list of possible values.

6. Click the **Apply the Time Window** macro button. The worksheets in the Statistics, Line Chart, and Stock Chart worksheets now show the month results.

7. Click the **Line Chart** macro button to go to the Line Chart worksheet. The chart shows the closing values and volume of the current month. See [Figure C-24](#).

Figure C-24Line Chart worksheet for the current month
8. View the **Stock Chart** and **Statistics** worksheets, and verify that they summarize the stock values for the current month.

Next you'll switch the time window to the current quarter.

9. Go to the **Report Control** worksheet, select **QUARTER** in cell C5, and then click the **Apply the Time Window** button to generate the quarterly reports.

10. Go to the **Statistics, Line Chart**, and **Stock Chart** worksheets and verify that their content changed to show the quarterly data.

11. Save the workbook, and then close it.

The Stock Report workbook is complete. Using the Visual Basic Editor, you created macros so that users can easily switch between worksheets and change the time window.

**Insight**

**Sharing Macros between Workbooks**

If you need to use the same macro for different workbooks, you can save time by sharing the macro in a special workbook containing your personal macros. The personal macro workbook has the filename Personal.xlsm and is stored in the C:\Users\username\AppData\Roaming\Microsoft\Excel\XLSTART folder where *username* is
your user name. This is a hidden folder. To access it directly, you must set Windows to display hidden files and folders. You do not need to do so because Excel will automatically open the Personal.xlsb file each time you start Excel, making all the macros stored in that file available to any open workbooks.

To record a personal macro and create the Personal.xlsb file if it is not already created, use the macro recorder and choose Personal Macro Workbook from the Store macro list box in the Record Macro dialog box, and then record the macro steps. After you have created one macro in the Personal.xlsb file, you can use the Visual Basic Editor to revise the macro and create other sub procedures. You can also copy the Personal.xlsb file from one computer to another to share your macros with other users.