Getting Started with Excel Macros

You need not be a power user to create and use simple VBA macros. Even casual users can simply turn on Excel's macro recorder.

Recording a macro is like programming a phone number into your cell phone. You first manually dial and save a number. Then when you want, you can redial those numbers with the touch of a button. Just as with numbers on a cell phone, you can record your actions in Excel while you perform them. While you record, Excel gets busy in the background, translating your keystrokes and mouse clicks to written VBA code. After you've recorded a macro, you can play back those actions anytime you wish.

This Part serves as an introduction to Excel Macros and VBA in general. Here, we give you a concise summary of how VBA works with Excel, giving you the basic foundation you need to work with the 101 macros listed in this book.

Becoming Familiar with Macro Recording Basics

To start recording your first macro, you need to first find the Macro Recorder, which is on the Developer tab. Unfortunately, Excel comes out of the box with the Developer tab hidden — you may not see it on your version of Excel at first. If you plan to work with VBA macros, you'll want to make sure that the Developer tab is visible. To display this tab

- 1. Choose Office→Excel Options.
- 2. In the Excel Options dialog box, select Customize Ribbon.
- 3. In the list box on the right, place a check mark next to Developer.
- 4. Click OK to return to Excel.

Now that you have the Developer tab showing in the Excel Ribbon, you can start up the Macro Recorder by selecting Record Macro from the Developer tab. This activates the Record Macro dialog box, as shown in Figure 1-1.

Rec	ord Macro 🤶 🏹
Mac	ro name:
	Macro1
Sho	rtcut <u>k</u> ey:
	Ctrl+
Stor	re macro in:
	This Workbook
Des	cription:
	OK Cancel

Figure 1-1: The Record Macro dialog box.

Here are the four parts of the Record Macro dialog box:

- Macro Name: This should be self-explanatory. Excel gives a default name to your macro, such as Macro1, but you should give your macro a name more descriptive of what it actually does. For example, you might name a macro that formats a generic table as FormatTable.
- Shortcut Key: Every macro needs an *event*, or something to happen, for it to run. This event can be a button press, a workbook opening, or in this case, a keystroke combination. When you assign a shortcut key to your macro, entering that combination of keys triggers your macro to run. This is an optional field.
- Store Macro In: This Workbook is the default option. Storing your macro in This Workbook simply means that the macro is stored along with the active Excel file. The next time you open that particular workbook, the macro is available to run. Similarly, if you send the workbook to another user, that user can run the macro as well (provided the macro security is properly set by your user more on that later in this Part, in the section titled "Looking at Other Macro Recording Concepts").
- Description: This is an optional field, but it can come in handy if you have numerous macros in a spreadsheet or if you need to give a user a more detailed description about what the macro does.

With the Record Macro dialog box open, follow these steps to create a simple macro that enters your name into a worksheet cell:

- 1. Enter a new single-word name for the macro to replace the default Macro1 name. A good name for this example is MyName.
- **2.** Assign this macro to the shortcut key Ctrl+Shift+N by entering uppercase N in the edit box labeled Shortcut Key.
- **3.** Click OK to close the Record Macro dialog box and begin recording your actions.

- **4.** Select any cell on your Excel spreadsheet, type your name into the selected cell, and then press Enter.
- Choose Developer→Code→Stop Recording (or click the Stop Recording button in the status bar).

Examining the macro

The macro was recorded in a new module named Module1. To view the code in this module, you must activate the Visual Basic Editor. You can activate the VB Editor in either of two ways:

- > Press Alt+F11.
- ➤ Choose Developer→Code→Visual Basic.

In the VB Editor, the Project window displays a list of all open workbooks and add-ins. This list is displayed as a tree diagram, which you can expand or collapse. The code that you recorded previously is stored in Module1 in the current workbook. When you double-click Module1, the code in the module appears in the Code window.

The macro should look something like this:

```
Sub MyName()
'
' MyName Macro
'
' Keyboard Shortcut: Ctrl+Shift+N
'
        ActiveCell.FormulaR1C1 = "Michael Alexander"
End Sub
```

The macro recorded is a Sub procedure that is named MyName. The statements tell Excel what to do when the macro is executed.

Notice that Excel inserted some comments at the top of the procedure. These comments are some of the information that appeared in the Record Macro dialog box. These comment lines (which begin with an apostrophe) aren't really necessary, and deleting them has no effect on how the macro runs. If you ignore the comments, you'll see that this procedure has only one VBA statement:

ActiveCell.FormulaR1C1 = "Michael Alexander"

This single statement causes the name you typed while recording to be inserted into the active cell.

Testing the macro

Before you recorded this macro, you set an option that assigned the macro to the Ctrl+Shift+N shortcut key combination. To test the macro, return to Excel by using either of the following methods:

- > Press Alt+F11.
- > Click the View Microsoft Excel button on the VB Editor toolbar.

When Excel is active, activate a worksheet. (It can be in the workbook that contains the VBA module or in any other workbook.) Select a cell and press Ctrl+Shift+N. The macro immediately enters your name into the cell.



In the preceding example, notice that you selected the cell to be formatted *before* you started recording your macro. This step is important. If you select a cell while the macro recorder is turned on, the actual cell that you selected will be recorded into the macro. In such a case, the macro would always format that particular cell, and it would not be a general-purpose macro.

Editing the macro

After you record a macro, you can make changes to it (although you must know what you're doing). For example, assume that you want your name to be bold. You could re-record the macro, but this modification is simple, so editing the code is more efficient. Press Alt+F11 to activate the VB Editor window. Then activate Module1 and insert the following statement before the End Sub statement:

```
ActiveCell.Font.Bold = True
```

The edited macro appears as follows:

```
Sub MyName()
'
' MyName Macro
'
' Keyboard Shortcut: Ctrl+Shift+N
'
ActiveCell.Font.Bold = True
ActiveCell.FormulaR1C1 = "Michael Alexander"
End Sub
```

Test this new macro, and you see that it performs as it should.

Comparing Absolute and Relative Macro Recording

Now that you've read about the basics of the Macro Recorder interface, it's time to go deeper and begin recording macros. The first thing you need to understand before you begin is that Excel has two modes for recording — absolute reference and relative reference.

Recording macros with absolute references

Excel's default recording mode is in absolute reference. As you may know, the term *absolute reference* is often used in the context of cell references found in formulas. When a cell reference in a formula is an absolute reference, it does not automatically adjust when the formula is pasted to a new location.

The best way to understand how this concept applies to macros is to try it out. Open the Chapter 1 Sample File.xlsx file and record a macro that counts the rows in the Branch list worksheet. (See Figure 1-2.)

	A	В	С	D	E	F	G	Н		
1		Region	Market	Branch			Region	Market	Branch	
2		NORTH	BUFFALO	601419			SOUTH	CHARLOTTE	173901	
3		NORTH	BUFFALO	701407			SOUTH	CHARLOTTE	301301	
4		NORTH	BUFFALO	802202			SOUTH	CHARLOTTE	302301	
5		NORTH	CANADA	910181			SOUTH	CHARLOTTE	601306	
6		NORTH	CANADA	920681			SOUTH	DALLAS	202600	
7		NORTH	MICHIGAN	101419			SOUTH	DALLAS	490260	
8		NORTH	MICHIGAN	501405			SOUTH	DALLAS	490360	
9		NORTH	MICHIGAN	503405			SOUTH	DALLAS	490460	
10		NORTH	MICHIGAN	590140			SOUTH	FLORIDA	301316	
11		NORTH	NEWYORK	801211			SOUTH	FLORIDA	701309	
12		NORTH	NEWYORK	802211			SOUTH	FLORIDA	702309	
13		NORTH	NEWYORK	804211			SOUTH	NEWORLEANS	601310	
14		NORTH	NEWYORK	805211			SOUTH	NEWORLEANS	602310	
15		NORTH	NEWYORK	806211			SOUTH	NEWORLEANS	801607	
40										

Figure 1-2: Your pre-totaled worksheet containing two tables.



The sample dataset used in this Part can be found on this book's companion website. See this book's Introduction for more on the companion website.

Follow these steps to record the macro:

- 1. Before recording, make sure cell A1 is selected.
- 2. Select Record Macro from the Developer tab.

- **3.** Name the macro **AddTotal**.
- 4. Choose This Workbook for the save location.
- 5. Click OK to start recording.

At this point, Excel is recording your actions. While Excel is recording, perform the following steps:

- 6. Select cell A16 and type Total in the cell.
- Select the first empty cell in Column D (D16) and type = COUNTA(D2:D15). This gives
 a count of branch numbers at the bottom of column D. You need to use the COUNTA
 function because the branch numbers are stored as text.
- 8. Click Stop Recording on the Developer tab to stop recording the macro.

	A	В	С	D	E	F	G	Н		
1		Region	Market	Branch			Region	Market	Branch	
2		NORTH	BUFFALO	601419			SOUTH	CHARLOTTE	173901	
3		NORTH	BUFFALO	701407			SOUTH	CHARLOTTE	301301	
4		NORTH	BUFFALO	802202			SOUTH	CHARLOTTE	302301	
5		NORTH	CANADA	910181			SOUTH	CHARLOTTE	601306	
6		NORTH	CANADA	920681			SOUTH	DALLAS	202600	
7		NORTH	MICHIGAN	101419			SOUTH	DALLAS	490260	
8		NORTH	MICHIGAN	501405			SOUTH	DALLAS	490360	
9		NORTH	MICHIGAN	503405			SOUTH	DALLAS	490460	
10		NORTH	MICHIGAN	590140			SOUTH	FLORIDA	301316	
11		NORTH	NEWYORK	801211			SOUTH	FLORIDA	701309	
12		NORTH	NEWYORK	802211			SOUTH	FLORIDA	702309	
13		NORTH	NEWYORK	804211			SOUTH	NEWORLEANS	601310	
14		NORTH	NEWYORK	805211			SOUTH	NEWORLEANS	602310	
15		NORTH	NEWYORK	806211			SOUTH	NEWORLEANS	801607	
16	Total			14						

The formatted worksheet should look something like the one in Figure 1-3.

Figure 1-3: Your post-totaled worksheet.

To see your macro in action, delete the total row you just added and play back your macro by following these steps:

- 1. Select Macros from the Developer tab.
- 2. Find and select the AddTotal macro you just recorded.
- **3.** Click the Run button.

If all goes well, the macro plays back your actions to a T and gives your table a total. Now here's the thing. No matter how hard you try, you can't make the AddTotal macro work on the second table. Why? Because you recorded it as an absolute macro.

To understand what this means, examine the underlying code. To examine the code, select Macros from the Developer tab to get the Macro dialog box you see in Figure 1-4.

Macro	?×
Macro name:	
AddTotal	Run
AddTotal	Step Into
	Edit
	Create
	Delete
×	Options
Macros in: All Open Workbooks	
	Cancel

Figure 1-4: The Excel Macro dialog box.

Select the AddTotal macro and click the Edit button. This opens the Visual Basic Editor to show you the code that was written when you recorded your macro:

```
Sub AddTotal()
Range("A16").Select
ActiveCell.FormulaR1C1 = "Total"
Range("D16").Select
ActiveCell.FormulaR1C1 = "=COUNTA(R[-14]C:R[-1]C)"
End Sub
```

Pay particular attention to lines two and four of the macro. When you asked Excel to select cell range A16 and then D16, those cells are exactly what it selected. Because the macro was recorded in absolute reference mode, Excel interpreted your range selection as absolute. In other words, if you select cell A16, that cell is what Excel gives you. In the next section, you take a look at what the same macro looks like when recorded in relative reference mode.

Recording macros with relative references

In the context of Excel macros, *relative* means relative to the currently active cell. So you should use caution with your active cell choice — both when you record the relative reference macro and when you run it.

First, make sure the Chapter 1 Sample File.xlsx file is open. (This file is available on this book's companion website.) Then, use the following steps to record a relative reference macro:

- 1. Select the Use Relative References option from the Developer tab, as shown in Figure 1-5.
- 2. Before recording, make sure cell A1 is selected.

- 3. Select Record Macro from the Developer tab.
- 4. Name the macro AddTotalRelative.
- 5. Choose This Workbook for the save location.
- 6. Click OK to start recording.
- 7. Select cell A16 and type Total in the cell.
- 8. Select the first empty cell in Column D (D16) and type = COUNTA(D2:D15).
- 9. Click Stop Recording on the Developer tab to stop recording the macro.

File	Ho	me	Insert	Page Layo	out	Developer
Visual Basic	Macros	E Re	cord Macro e Relative F acro Securit	References		Source R
		Co	de			- A. 161-1015

Figure 1-5: Recording a macro with relative references.

At this point, you have recorded two macros. Take a moment to examine the code for your newly-created macro.

Select Macros from the Developer tab to open the Macro dialog box. Here, choose the AddTotalRelative macro and click Edit.

Again, this opens the Visual Basic Editor to show you the code that was written when you recorded your macro. This time, your code looks something like the following:

```
Sub AddTotalRelative()
ActiveCell.Offset(15, 0).Range("A1").Select
ActiveCell.FormulaR1C1 = "Total"
ActiveCell.Offset(0, 3).Range("A1").Select
ActiveCell.FormulaR1C1 = "=COUNTA(R[-14]C:R[-1]C)"
End Sub
```

Notice that there are no references to any specific cell ranges at all (other than the starting point "A1"). Let's take a moment to take a quick look at what the relevant parts of this VBA code really mean.

Notice that in line 2, Excel uses the Offset property of the active cell. This property tells the cursor to move a certain number of cells up or down and a certain number of cells left or right.

The Offset property code tells Excel to move 15 rows down and 0 columns across from the active cell (in this case, A1). There's no need for Excel to explicitly select a cell as it did when recording an absolute reference macro.

To see this macro in action, delete the total row for both tables and do the following:

- 1. Select cell A1.
- 2. Select Macros from the Developer tab.
- **3.** Find and select the AddTotalRelative macro.
- 4. Click the Run button.
- 5. Now select cell F1.
- 6. Select Macros from the Developer tab.
- 7. Find and select the AddTotalRelative macro.
- 8. Click the Run button.

Notice that this macro, unlike your previous macro, works on both sets of data. Because the macro applies the totals *relative* to the currently active cell, the totals are applied correctly.

For this macro to work, you simply need to ensure that

- > You've selected the correct starting cell before running the macro.
- The block of data has the same number of rows and columns as the data on which you recorded the macro.

Hopefully, this simple example has given you a firm grasp of macro recording with both absolute and relative references.

Looking at Other Macro Recording Concepts

At this point, you should feel comfortable recording your own Excel Macros. Now here are some of the other important concepts you'll need to keep in mind when working with macros.

Macro-enabled file extensions

Beginning with Excel 2007, Excel has a separate file extension for workbooks that contain macros.

You see, Excel 2010 workbooks have the standard file extension .xlsx. Files with the .xlsx extension cannot contain macros. If your workbook contains macros and you then save that workbook as an .xlsx file, your macros are removed automatically. Excel warns you that macro content will be disabled when saving a workbook with macros as an .xlsx file.

If you want to retain the macros, you must save your file as an *Excel Macro-Enabled Workbook*. This gives your file an .xlsm extension. The idea is that all workbooks with an .xlsx file extension are automatically known to be safe, whereas you can recognize .xlsm files as a potential threat.

Macro security in Excel 2010

With the release of Office 2010, Microsoft introduced significant changes to its Office security model. One of the most significant changes is the concept of trusted documents. Without getting into the technical minutia, a *trusted document* is essentially a workbook you have deemed safe by enabling macros.

If you open a workbook that contains macros in Excel 2010, you see a yellow bar message under the Ribbon stating that macros (active content) have in effect, been disabled.

If you click Enable, it automatically becomes a trusted document. This means you no longer are prompted to enable the content as long as you open that file on your computer. The basic idea is that if you told Excel that you "trust" a particular workbook by enabling macros, it is highly likely that you will enable macros each time you open it. Thus, Excel remembers that you've enabled macros before and inhibits any further messages about macros for that workbook.

This is great news for you and your clients. After enabling your macros just one time, they won't be annoyed at the constant messages about macros, and you won't have to worry that your macro-enabled dashboard will fall flat because macros have been disabled.

Trusted locations

If the thought of any macro message coming up (even one time) unnerves you, you can set up a trusted location for your files. A *trusted location* is a directory that is deemed a safe zone where only trusted workbooks are placed. A trusted location allows you and your clients to run a macro-enabled workbook with no security restrictions as long as the workbook is in that location.

To set up a trusted location, follow these steps:

1. Select the Macro Security button on the Developer tab.

This activates the Trust Center dialog box.

- **2.** Click the Trusted Locations button. This opens the Trusted Locations menu (see Figure 1-6), which shows you all the directories that are considered trusted.
- 3. Click the Add New Location button.
- 4. Click Browse to find and specify the directory that will be considered a trusted location.

After you specify a trusted location, any Excel file that is opened from this location will have macros automatically enabled.

Trusteu Publistiers	Trusted Location	s		
Trusted Locations	Warning: All these	locations are treated as trusted s	ources for opening files. If you change or add a loc	ation, make sure th
Trusted Documents	the new location i	s secure.		action, mance sure th
	Path		Description	Date Modified 7
Add-ins	User Locations			
ActiveX Settings	C:\Program Files\I	dicrosoft Office\Templates\	Excel 2010 default location: Application Templates	
Actives seconds	C:\an\Applicatio	in Data\Microsoft\Excel\XLSTART\	Excel 2010 default location: User StartUp	
Macro Settings	C:\gram Files\M	icrosoft Office\Office14\XLSTART\	Excel 2010 default location: Excel StartUp	
	C:\alexan\Applie	ation Data\Microsoft\Templates\	Excel 2010 default location: User Templates	
Protected View	C:\ram Files\Mic	rosoft Office\Office14\STARTUP\	Excel 2010 default location: Office StartUp	
Message Bar	C:\Program Files\I	/licrosoft Office\Office14\Library\	Excel 2010 default location: Add-ins	
External Content	Policy Location	s		
File Block Settings	Path:	C:\Program Files\Microsoft Office	e\Templates\	
	Description:	Excel 2010 default location: Appl	ication Templates	
Privacy Options				
	Date Modified:			
	Sub Folders:	Allowed		
			Add new location Remov	Modify
	_			<u>m</u> ouny
	Allo <u>w</u> Trusted	Locations on my network (not reco	ommended)	

Figure 1-6: The Trusted Locations menu allows you to add directories that are considered trusted.

Storing macros in your Personal Macro Workbook

Most user-created macros are designed for use in a specific workbook, but you may want to use some macros in all your work. You can store these general-purpose macros in the Personal Macro Workbook so that they're always available to you. The Personal Macro Workbook is loaded whenever you start Excel. This file, named personal.xlsb, doesn't exist until you record a macro using Personal Macro Workbook as the destination.



The Personal Macro Workbook normally is in a hidden window to keep it out of the way.

To record the macro in your Personal Macro Workbook, select the Personal Macro Workbook option in the Record Macro dialog box before you start recording. This option is in the Store Macro In drop-down list.

If you store macros in the Personal Macro Workbook, you don't have to remember to open the Personal Macro Workbook when you load a workbook that uses macros. When you want to exit, Excel asks whether you want to save changes to the Personal Macro Workbook.

Assigning a macro to a button and other form controls

When you create macros, you may want to have a clear and easy way to run each macro. A basic button can provide a simple but effective user interface.

As luck would have it, Excel offers a set of *form controls* designed specifically for creating user interfaces directly on spreadsheets. There are several different types of form controls, from buttons (the most commonly used control) to scrollbars.

The idea behind using a form control is simple. You place a form control on a spreadsheet and then assign a macro to it — that is, a macro you've already recorded. When a macro is assigned to the control, that macro is executed, or *played*, when the control is clicked.

Take a moment to create a button for the AddTotalRelative macro you created earlier. Here's how:

- 1. Click the Insert button under the Developer tab. (See Figure 1-7.)
- 2. Select the Button Form Control from the drop-down list that appears.
- **3.** Click the location where you want to place your button. When you drop the button control onto your spreadsheet, the Assign Macro dialog box, as shown in Figure 1-8, activates and asks you to assign a macro to this button.
- 4. Select the macro you want to assign to the button and then click OK.



Figure 1-7: You can find the form controls in the Developer tab.

Assign Macro		?×
Macro name:		
AddTotalRelative	E	Edit
AddTotal		
AddTotalRelative		Record,
	~	
Macros in: All Open Workbooks	~	
Description		
	ОК	Capcel

Figure 1-8: Assign a macro to the newly-added button.

Form controls versus ActiveX controls

Notice the form controls and ActiveX controls in Figure 1-7. Although they look similar, they're quite different. Form controls are designed specifically for use on a spreadsheet, and ActiveX controls are typically used on Excel user forms. As a general rule, you should always use form controls when working on a spreadsheet. Why? Form controls need less overhead, so they perform better, and configuring form controls is far easier than configuring their ActiveX counterparts.

At this point, you have a button that runs your macro when you click it! Keep in mind that all the controls in the Form Controls group (shown in Figure 1-7) work in the same way as the command button, in that you assign a macro to run when the control is selected.

Placing a macro on the Quick Access toolbar

You can also assign a macro to a button in Excel's Quick Access toolbar:

- 1. Right-click your Quick Access toolbar and select Customize Quick Access Toolbar.
- 2. Click the Quick Access Toolbar button on the left of the Excel Options dialog box.
- 3. Select Macros from the drop-down list on the left.
- 4. Select the macro you want to add and click the Add button.
- 5. Change the icon by clicking the Modify button.

Working in the Visual Basic Editor

The Visual Basic Editor (VBE) is a separate application where you write and edit your VBA macros. You can't run the VBE separately; Excel must be running in order for the VBE to run.

Activating the VBE

The quickest way to activate the VBE is to press Alt+F11 when Excel is active. To return to Excel, press Alt+F11 again.

You can also activate the VBE by using the Developer→Code→Visual Basic command.

Understanding VBE components

Figure 1-9 shows the VBE program with some of the key parts identified. Because so much is going on in the VBE, I like to maximize the window to see as much as possible.

Chances are your VBE program window won't look exactly like what you see in Figure 1-9. The VBE contains several windows and is highly customizable. You can hide windows, rearrange windows, dock windows, and so on.



Figure 1-9: The VBE with significant elements identified.

Menu bar

The VBE menu bar works just like every other menu bar you've encountered. It contains commands that you use to do things with the various components in the VBE. You will also find that many of the menu commands have shortcut keys associated with them.

The VBE also features shortcut menus. You can right-click virtually anything in the VBE and get a shortcut menu of common commands.

Toolbar

The Standard toolbar, which is directly under the menu bar by default, is one of four VBE toolbars available. You can customize the toolbars, move them around, display other toolbars, and so

on. If you're so inclined, use the View→Toolbars command to work with VBE toolbars. Most people just leave them as they are.

Project window

The Project window displays a tree diagram that shows every workbook currently open in Excel (including add-ins and hidden workbooks). Double-click items to expand or contract them. We discuss this window in more detail in the "Working with the Project Window" section later in this Part.

If the Project window is not visible, press Ctrl+R or use the View→Project Explorer command. To hide the Project window, click the Close button in its title bar. Alternatively, right-click anywhere in the Project window and select Hide from the shortcut menu.

Code window

A Code window contains VBA code. Every object in a project has an associated Code window. To view an object's Code window, double-click the object in the Project window. For example, to view the Code window for the Sheet1 object, double-click Sheet1 in the Project window. Unless you've added some VBA code, the Code window will be empty.

You find out more about Code windows later in this Part's "Working with a Code Window" section.

Immediate window

The Immediate window may or may not be visible. If it isn't visible, press Ctrl+G or use the View→ Immediate Window command. To close the Immediate window, click the Close button in its title bar (or right-click anywhere in the Immediate window and select Hide from the shortcut menu).

The Immediate window is most useful for executing VBA statements directly and for debugging your code. If you're just starting out with VBA, this window won't be all that useful, so feel free to hide it and free up some screen space for other things.

Working with the Project window

When you're working in the VBE, each Excel workbook and add-in that's open is a project. You can think of a *project* as a collection of objects arranged as an outline. You can expand a project by clicking the plus sign (+) at the left of the project's name in the Project window. Contract a project by clicking the minus sign (-) to the left of a project's name. Or, you can double-click the items to expand and contract them.

Figure 1-10 shows a Project window with two projects listed: a workbook named Book1 and a workbook named Book2.



Figure 1-10: This Project window lists two projects. They are expanded to show their objects.

Every project expands to show at least one *node* called Microsoft Excel Objects. This node expands to show an item for each sheet in the workbook (each sheet is considered an object), and another object called ThisWorkbook (which represents the Workbook object). If the project has any VBA modules, the project listing also shows a Modules node.

Adding a new VBA module

When you record a macro, Excel automatically inserts a VBA module to hold the recorded code. The workbook that holds the module for the recorded macro depends on where you chose to store the recorded macro, just before you started recording.

In general, a VBA module can hold three types of code:

- Declarations: One or more information statements that you provide to VBA. For example, you can declare the data type for variables you plan to use, or set some other module-wide options.
- Sub procedures: A set of programming instructions that performs some action. All recorded macros will be Sub procedures.
- Function procedures: A set of programming instructions that returns a single value (similar in concept to a worksheet function, such as Sum).

A single VBA module can store any number of Sub procedures, Function procedures, and declarations. How you organize a VBA module is completely up to you. Some people prefer to keep all their VBA code for an application in a single VBA module; others like to split up the code into several different modules. It's a personal choice, just like arranging furniture.

Follow these steps to manually add a new VBA module to a project:

- 1. Select the project's name in the Project window.
- 2. Choose Insert→Module.

Or you can

- 1. Right-click the project's name.
- **2.** Choose Insert \rightarrow Module from the shortcut menu.

The new module is added to a Modules folder in the Project window (see Figure 1-11). Any modules you create in a given workbook are placed in this Modules folder.



Figure 1-11: Code modules are visible in the Project window in a folder called Modules.

Removing a VBA module

You may want to remove a code module that is no longer needed. To do so, follow these steps:

- 1. Select the module's name in the Project window.
- **2.** Choose File \rightarrow Remove *xxx*, where *xxx* is the module name.

Or

- 1. Right-click the module's name.
- 2. Choose Remove *xxx* from the shortcut menu.



Excel, always trying to keep you from doing something you'll regret, asks if you want to export the code in the module before you delete it. Almost always, you don't. (If you do want to export the module, see the next section, "Working with a Code Window.")

You can remove VBA modules, but there is no way to remove the other code modules — those for the Sheet objects or ThisWorkbook.

Working with a Code Window

As you become proficient with VBA, you spend lots of time working in Code windows. Macros that you record are stored in a module, and you can type VBA code directly into a VBA module.

Minimizing and maximizing windows

Code windows are much like workbook windows in Excel. You can minimize them, maximize them, resize them, hide them, rearrange them, and so on. Most people find it much easier to maximize the Code window that they're working on. Doing so lets you see more code and keeps you from getting distracted.

To maximize a Code window, click the maximize button in its title bar (right next to the X). Or, just double-click its title bar to maximize it. To restore a Code window to its original size, click the Restore button. When a window is maximized, its title bar isn't really visible, so you'll find the Restore button to the right of the Type a Question for Help box.

Sometimes, you may want to have two or more Code windows visible. For example, you may want to compare the code in two modules or copy code from one module to another. You can arrange the windows manually, or use the Window→Tile Horizontally or Window→Tile Vertically commands to arrange them automatically.

You can quickly switch among code windows by pressing Ctrl+Tab. If you repeat that key combination, you keep cycling through all the open code windows. Pressing Ctrl+Shift+Tab cycles through the windows in reverse order.

Minimizing a Code window gets it out of the way. You can also click the window's Close button in a Code window's title bar to close the window completely. (Closing a window just hides it; you won't lose anything.) To open it again, just double-click the appropriate object in the Project window. Working with these Code windows *sounds* more difficult than it really is.

Getting VBA code into a module

Before you can do anything meaningful, you must have some VBA code in the VBA module. You can get VBA code into a VBA module in three ways:

- > Use the Excel macro recorder to record your actions and convert them to VBA code
- > Enter the code directly
- > Copy the code from one module and paste it into another

You have discovered the excellent method for creating code by using the Excel Macro recorder. However, not all tasks can be translated to VBA by recording a macro. You often have to enter your code directly into the module. Entering code directly basically means either typing the code yourself or copying and pasting code you have found somewhere else.

Entering and editing text in a VBA module works as you might expect. You can select, copy, cut, paste, and do other things to the text.

A single line of VBA code can be as long as you like. However, you may want to use the linecontinuation character to break up lengthy lines of code. To continue a single line of code (also known as a *statement*) from one line to the next, end the first line with a space followed by an underscore (_). Then continue the statement on the next line. Here's an example of a single statement split into three lines:

```
Selection.Sort Key1:=Range("A1"), _
Order1:=xlAscending, Header:=xlGuess, _
Orientation:=xlTopToBottom
```

This statement would perform exactly the same way if it were entered in a single line (with no line-continuation characters). Notice that the second and third lines of this statement are indented. Indenting is optional, but it helps clarify the fact that these lines are not separate statements.

The VBE has multiple levels of undo and redo. If you deleted a statement that you shouldn't have, use the Undo button on the toolbar (or press Ctrl+Z) until the statement appears again. After undoing, you can use the Redo button to perform the changes you've undone.

Ready to enter some real, live code? Try the following steps:

- 1. Create a new workbook in Excel.
- 2. Press Alt+F11 to activate the VBE.
- 3. Click the new workbook's name in the Project window.
- 4. Choose Insert→Module to insert a VBA module into the project.

5. Type the following code into the module:

```
Sub GuessName()
Dim Msg as String
Dim Ans As Long
Msg = "Is your name " & Application.UserName & "?"
Ans = MsgBox(Msg, vbYesNo)
If Ans = vbNo Then MsgBox "Oh, never mind."
If Ans = vbYes Then MsgBox "I must be clairvoyant!"
End Sub
```

6. Make sure the cursor is located anywhere within the text you typed and press F5 to execute the procedure.



F5 is a shortcut for the Run→Run Sub/UserForm command.

When you enter the code listed in Step 5, you might notice that the VBE makes some adjustments to the text you enter. For example, after you type the Sub statement, the VBE automatically inserts the End Sub statement. And if you omit the space before or after an equal sign, the VBE inserts the space for you. Also, the VBE changes the color and capitalization of some text. This is all perfectly normal. It's just the VBE's way of keeping things neat and readable.

If you followed the previous steps, you just created a VBA sub procedure, also known as a *macro*. When you press F5, Excel executes the code and follows the instructions. In other words, Excel evaluates each statement and does what you told it to do. You can execute this macro any number of times — although it tends to lose its appeal after a few dozen executions.

This simple macro uses the following concepts:

- > Defining a Sub procedure (the first line)
- Declaring variables (the Dim statements)
- Assigning values to variables (Msg and Ans)
- > Concatenating (joining) a string (using the & operator)
- ► Using a built-in VBA function (MsgBox)
- Using built-in VBA constants (vbYesNo, vbNo, and vbYes)
- ► Using an If-Then construct (twice)
- Ending a Sub procedure (the last line)

As we mentioned previously, you can copy and paste code into a VBA module. For example, a Sub or Function procedure that you write for one project might also be useful in another

project. Instead of wasting time reentering the code, you can activate the module and use the normal copy-and-paste procedures (Ctrl+C to copy and Ctrl+V to paste). After pasting it into a VBA module, you can modify the code as necessary.

Customizing the VBA Environment

If you're serious about becoming an Excel programmer, you'll spend a lot of time with VBA modules on your screen. To help make things as comfortable as possible, the VBE provides quite a few customization options.

When the VBE is active, choose Tools-Options. You'll see a dialog box with four tabs: Editor, Editor Format, General, and Docking. Take a moment to explore some of the options found on each tab.

The Editor tab

Figure 1-12 shows the options accessed by clicking the Editor tab of the Options dialog box. Use the option in the Editor tab to control how certain things work in the VBE.

Options	$\overline{\mathbf{X}}$
Editor Editor Format General Dock	ng
Code Settings	
🔽 Auto Syntax Check	Auto Indent
 Require Variable Declaration Auto List Members 	Tab Width: 4
🔽 Auto Quick Info	
🔽 Auto Data Tips	
Window Settings	
🔽 Drag-and-Drop Text Editing	
✓ Default to Full Module View	
Procedure Separator	
0	K Cancel Help

Figure 1-12: The Editor tab in the Options dialog box.

The Auto Syntax Check option

The Auto Syntax Check setting determines whether the VBE pops up a dialog box if it discovers a syntax error while you're entering your VBA code. The dialog box tells roughly what the problem is. If you don't choose this setting, VBE flags syntax errors by displaying them in a different color from the rest of the code, and you don't have to deal with any dialog boxes popping up on your screen.



I usually keep this setting turned off because I think the dialog boxes are annoying and I can usually figure out what's wrong with a statement. Before I was a VBA veteran, I found this setting quite helpful.

The Require Variable Declaration option

If the Require Variable Declaration option is set, VBE inserts the following statement at the beginning of each new VBA module you insert:

Option Explicit

Changing this setting affects only new modules, not existing modules. If this statement appears in your module, you must explicitly define each variable you use. Using a Dim statement is one way to declare variables.

The Auto List Members option

If the Auto List Members option is set, VBE provides some help when you're entering your VBA code. It displays a list that would logically complete the statement you're typing. This is one of the best features of the VBE.

The Auto Quick Info option

If the Auto Quick Info option is selected, VBE displays information about functions and their arguments as you type. This is similar to the way Excel lists the arguments for a function as you start typing a new formula.

The Auto Data Tips option

If the Auto Data Tips option is set, VBE displays the value of the variable over which your cursor is placed when you're debugging code. This is turned on by default and often quite useful. You have no reason to turn this option off.

The Auto Indent setting

The Auto Indent setting determines whether VBE automatically indents each new line of code the same as the previous line. I'm big on using indentations in my code, so I keep this option on.



By the way, you should use the Tab key to indent your code, not the spacebar. Also, you can use Shift+Tab to "unindent" a line of code. If you want to indent more than just one line, select all lines you want to indent and then press the Tab key.

The VBE's Edit toolbar (which is hidden by default) contains two useful buttons: Indent and Outdent. These buttons let you quickly indent or "unindent" a block of code. Select the code and click one of these buttons to change the block's indenting.

The Drag-and-Drop Text Editing option

The Drag-and-Drop Text Editing option, when enabled, lets you copy and move text by dragging and dropping with your mouse. I keep this option turned on, but I never use it. I prefer to copy and move by using the keyboard.

The Default to Full Module View option

The Default to Full Module View option sets the default state for new modules. (It doesn't affect existing modules.) If set, procedures in the Code window appear as a single scrollable list. If this option is turned off, you can see only one procedure at a time. I keep this option turned on.

The Procedure Separator option

When the Procedure Separator option is turned on, separator bars appear at the end of each procedure in a Code window. I like the idea of separator bars, so I keep this option turned on.

The Editor Format tab

Figure 1-13 shows the Editor Format tab of the Options dialog box. With this tab, you can customize the way the VBE looks.

Options	$\overline{\mathbf{X}}$
Editor Editor Format General Docking Code Colors Selection Text Syntax Error Text Breakpoint Text Breakpoint Text Comment Text Keyword Text Foreground: Background: Indicator: Auto Auto Auto Auto	Font: Courier New (Western) 💌 Size: 10 💌 Margin Indicator Bar Sample ABCXYZabcxyz
OK	Cancel Help

Figure 1-13: Change the VBE's looks with the Editor Format tab.

The Code Colors option

The Code Colors option lets you set the text color and background color displayed for various elements of VBA code. This is largely a matter of personal preference. Personally, I find the default colors to be just fine. But for a change of scenery, I occasionally play around with these settings.

The Font option

The Font option lets you select the font that's used in your VBA modules. For best results, stick with a fixed-width font such as Courier New. In a *fixed-width font*, all characters are exactly the same width. This makes your code more readable because the characters are nicely aligned vertically and you can easily distinguish multiple spaces (which is sometimes useful).

The Size setting

The Size setting specifies the point size of the font in the VBA modules. This setting is a matter of personal preference determined by your video display resolution and how good your eyesight is.

The Margin Indicator Bar option

This option controls the display of the vertical margin indicator bar in your modules. You should keep this turned on; otherwise, you won't be able to see the helpful graphical indicators when you're debugging your code.

The General tab

Figure 1-14 shows the options available under the General tab in the Options dialog box. In almost every case, the default settings are just fine.

The most important setting on the General tab is Error Trapping. We strongly suggest that you use the Break on Unhandled Errors setting. This ensures Excel can identify errors as you type your code.

Options 🛛 🕅				
Editor Editor Format General D	ocking			
Form Grid Settings	Edit and Continue			
🔽 Show Grid	Notify Before State Loss			
Grid Units: Points	Error Trapping			
Width: 6	C Break on All Errors			
Height: 6	C Break in Class Module			
Align Controls to Grid	Break on Unhandled Errors			
	Compile			
Show ToolTips	Compile On Demand			
Collapse Proj. Hides Windows	Background Compile			
	OK Cancel Help			

Figure 1-14: The General tab of the Options dialog box.

The Docking tab

Figure 1-15 shows the Docking tab. These options determine how the various windows in the VBE behave. When a window is *docked*, it is fixed in place along one of the edges of the VBE program window. This makes it much easier to identify and locate a particular window. If you turn off all docking, you have a big, confusing mess of windows. Generally, the default settings work fine.

Options 🔀
Editor Editor Format General Docking
Dockable
☑ Immediate Window
☑ Locals Window
☑ Watch Window
V Project Explorer
V Properties Window
Cobject Browser
OK Cancel Help

Figure 1-15: The Docking tab of the Options dialog box.

Discovering the Excel Object Model

In Excel, you deal with workbooks, worksheets, and ranges on a daily basis. You likely think of each of these "objects" as all part of Excel, not really separating them in your mind. However, Excel thinks about these internally as all part of a hierarchical model called the *Excel object model*. The Excel object model is a clearly defined set of objects that are structured according to the relationships between them.

In this section, we give you a brief overview of the object model, as well as some of the other terminology and concepts you will encounter in the upcoming 101 macros.

Understanding objects

In the real world, you can describe everything you see as an object. When you look at your house, it is an object. Your house has rooms; those rooms are also separate objects. Those rooms may have closets. Those closets are likewise objects. As you think about your house, the rooms, and the closets, you may see a hierarchical relationship between them. Excel works in the same way.

In Excel, the Application object is the all-encompassing object — similar to your house. Inside the Application object, Excel has a workbook. Inside a workbook is a worksheet. Inside that is a range. These are all objects that live in a hierarchical structure.

To point to a specific object in VBA, you can traverse the object model. For example, to get to cell A1 on Sheet 1, you can enter this code:

```
Application.Activeworbook.Sheets("Sheet1").Range("A1").Select
```

In most cases, the object model hierarchy is understood, so you don't have to type every level. Entering this code also gets you to cell A1 because Excel infers that you mean the active workbook, and the active sheet:

Range("A1").Select

Indeed, if you have you cursor already in cell A1, you can simply use the ActiveCell object, negating the need to actually spell out the range.

Activecell.Select

Understanding collections

Many of Excel's objects belong to collections. Your house sits within a neighborhood, for example, which is a collection of houses called a neighborhood. Each neighborhood sits in a collection of neighborhoods called a city. Excel considers collections to be objects themselves.

In each Workbook object, you have a collection of Worksheets. The Worksheets collection is an object that you can call upon through VBA. Each worksheet in your workbook lives in the Worksheets collection.

If you want to refer to a worksheet in the Worksheets collection, you can refer to it by its position in the collection, as an index number starting with 1, or by its name, as quoted text. If you run these two lines of code in a workbook that has only one worksheet called MySheet, they both do the same thing:

```
Worksheets(1).Select
Worksheets("MySheet").Select
```

If you have two worksheets in the active workbook that have the names MySheet and YourSheet, in that order, you can refer to the second worksheet by typing either of these statements:

Worksheets(2).Select Worksheets("YourSheet").Select

If you want to refer to a worksheet in a workbook called MySheet in a particular workbook that is not active, you must qualify the worksheet reference and the workbook reference, as follows:

Workbooks("MyData.xls").Worksheets("MySheet").Select

Understanding properties

Properties are essentially the characteristics of an object. Your house has a color, a square footage, an age, and so on. Some properties can be changed — like the color of your house. Other properties can't be changed — like the age of your house.

Likewise, an object in Excel like the Worksheet object has a sheet name property that can be changed, and a Rows.Count row property that cannot.

You refer to the property of an object by referring to the object, and then the property. For instance, you can change the name of your worksheet by changing its Name property.

In this example, you are renaming Sheet1 to MySheet:

Sheets("Sheet1").Name = "MySheet"

Some properties are read-only, which means that you can't assign a value to them directly — for instance, the Text property of cell. The Text property gives you the formatted appearance of value in a cell, but you cannot overwrite or change it.

Understanding methods

Methods are the actions that can be performed against an object. It helps to think of methods as verbs. You can paint your house, so in VBA, that translates to something like

house.paint

A simple example of an Excel method is the Select method of the Range object:

```
Range("A1").Select
```

Another is the Copy method of the Range object:

Range("A1").Copy

Some methods have parameters that can dictate how it is applied. For instance, the Paste method can be used more effectively by explicitly defining the Destination parameter.

```
ActiveSheet.Paste Destination:=Range("B1")
```

Taking a Brief Look at Variables

Another concept you will see throughout the macros in this book is the concept of variables. We need to dedicate a few words to this concept because it plays a big part in most of the macros you will encounter here.

You can think of variables as memory containers that you can use in your procedures. There are different types of variables, each tasked with holding a specific type of data.

Some of the common types of variables you will see in this book are

- > String: Holds textual data
- ▶ Integer: Holds numeric data ranging from -32,768 to 32,767
- ▶ Long: Holds numeric data ranging from -2,147,483,648 to 2,147,483,647
- > Double: Holds floating point numeric data
- > Variant: Holds any kind of data
- Boolean: Holds binary data that returns True or False
- > **Object:** Holds an actual object from the Excel object model

The term used for creating a variable in a macro is *declaring* a variable. You do so by entering Dim (an abbreviation for dimension), the name of your variable, and then the type. For instance:

Dim MyText as String Dim MyNumber as Integer Dim MyWorksheet as Worksheet

After you create your variable, you can fill it with data. Here are a few simple examples of how you would create a variable and assign a value to it:

```
Dim MyText as String
Mytext = Range("A1").Value
```

```
Dim MyNumber as Integer
MyNumber = Range("B1").Value * 25
```

```
Dim MyObject as Worksheet
Set MyWorksheet = Sheets("Sheet1")
```

The values you assign to your variables often come from data stored in your cells. However, the values may also be information that you yourself create. It all depends on the task at hand. This notion becomes clearer as you go through the macros in the book.

About the Macros in This Book

As we mention in the Introduction, the macros in this book are designed to get you up and running with VBA in the quickest way possible. Each macro tackles a common task that benefits from automation. The idea here is learning through application. This book is designed so that you can implement the macro, while getting a clear understanding of what the macro does and how it works.

Getting the sample files

Each macro in this book has an associated sample file. These sample files give you the ability to see the macro working, as well as the ability to review the code. You can also use the sample files to copy and paste the code into your environment (as opposed to typing each macro in from scratch). See the "About the Companion Website" section in this book's Introduction for complete instructions on how to download the sample code.

Using the sample files

Each macro in this book has detailed instructions on where to copy and paste the code. You should open the sample file associated with the macro, go to the Visual Basic Editor (by pressing Alt+F11), and then copy the code. After you've copied the code, you can go to your workbook, go into the Visual Basic Editor, and paste the code into the appropriate location.

Note that in some of the Macro examples in this book, you need to change some aspect of the macro to suit your situation. For instance, Macro 12 in Part II demonstrates how to open all the

Excel files in a directory. In that example, we point to the $C: \ensuremath{\mathbb{C}}$ directory. Before you use this particular macro, you need to edit that portion of the macro to point to *your* target directory.

If a macro is not working for you, it's probably because you need to change some component of the macro. Pay special attention to Range addresses, directory names, and any other hard-coded names. We built these hard-coded values into the macro for demonstration purposes; with the full intent that you, the reader, would edit the macro to alter these hard-coded names to fit your scenario.

Things to keep in mind

Here are some final things to keep in mind while working with the macros in this book:

- Macro-enabled file extensions: Remember that any file that contains a macro must have the .xlsm file extension. See the section called "Macro-enabled file extensions" in this Part for more details.
- Macro security: Keep in mind that Excel will not run macros until they are Enabled. As you implement these macros, it's important to understand the steps that you and your customers will need to take to comply with Excel's macro security measures. The section in this Part called "Macro security in Excel 2010" highlights these steps.
- > You cannot undo macro actions: When working in Excel, you can often undo the actions you have taken. This is because Excel keeps a log (called the *undo stack*) that records the last 100 actions you have taken. However, running a macro automatically destroys the undo stack, clearing the log of the actions you have taken. You must keep this in mind as you start writing and running your own macros. You cannot undo the action you take in a macro.
- Where to go from here: As mentioned before, these macros were designed to get you started with VBA. If you find a developing passion for Excel VBA, you may want to know where to get a more detailed reference on Excel VBA in general. Allow us to recommend *Excel 2010 Power Programming with VBA* by John Walkenbach (Wiley) as the next step in your learning. This reference is a comprehensive guide to VBA, diving deeper into the Excel Object model.