2. **In the VBE window, locate the window called Project.**

   The Project window (also known as the Project Explorer window) contains a list of all workbooks that are currently open. Each project is arranged as a *tree* and can be expanded (to show more information) or contracted (to show less information).

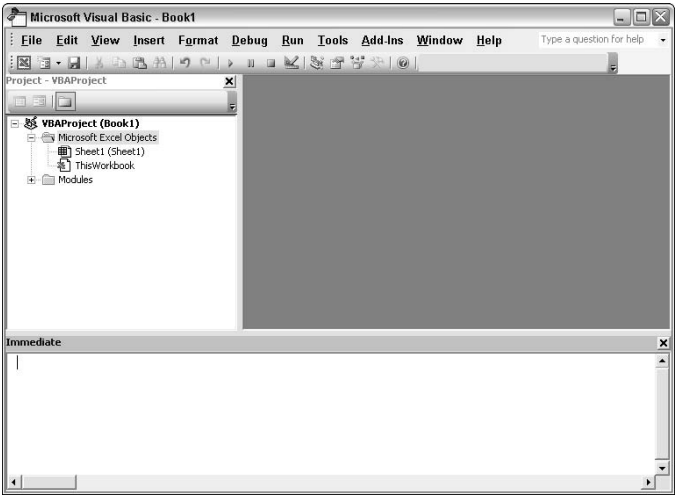


Figure 2-4: The Visual Basic Editor is where you view and edit VBA code.

The VBE uses quite a few different windows, any of which can be either open or closed. If a window isn't immediately visible in the VBE, you can choose an option from the View menu to display the window. For instance, if the Project window is not visible, you can choose View➪Project Explorer (or press Ctrl+R) to display it. You can display any other VBE window in a similar manner.

3. **Select the project that corresponds to the workbook in which you recorded the macro.**

   If you haven't saved the workbook, the project is probably called VBAProject (Book1).

4. **Click the plus sign (+) to the left of the folder named Modules.**

   The tree expands to show Module1, which is the only module in the project.

5. **Double-click Module1.**

   The VBA code in that module is displayed in a Code window. Figure 2-5 shows how it looks on my screen. Because the VBE program window is highly customizable, your screen may not look exactly the same.

The code in Module1 should look like this:

```
Sub ConvertFormulas()
```

```
'
' ConvertFormulas Macro
' Macro recorded 3/24/2004 by John Walkenbach
'
' Keyboard Shortcut: Ctrl+Shift+C
'
    Selection.Copy
    Selection.PasteSpecial Paste:=xlValues, _
        Operation:=xlNone, SkipBlanks:=False, _
        Transpose:=False
   Application.CutCopyMode = False
End Sub
```

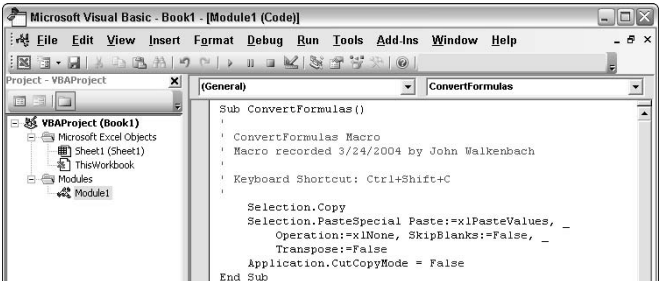
Notice that this listing looks just a bit different than what you see on your screen and from what is shown in Figure 2-5. (Take a look at the lines beginning with `Selection.PasteSpecial`.) The lines have simply been reformatted to fit on the printed page. The only difference is where each line of the macro is divided.

At this point, the macro probably looks like Greek to you. Don't worry. Travel a few chapters down the road and all will be as clear as the view from Olympus.

The ConvertFormulas macro (also known as a *Sub procedure*) consists of several statements. Excel executes the statements one by one, from top to bottom. A statement preceded by an apostrophe (') is a comment.

Comments are included only for your information and are essentially ignored. In other words, Excel doesn't execute comments.

The first actual VBA statement (which begins with the word *Sub*) identifies the macro as a Sub procedure and gives its name — you provided this name before you started recording the macro. The next statement tells Excel to copy the cells in the selection. The next statement corresponds to the options you selected in the Paste Special dialog box. (This statement occupies two lines, using VBA's line continuation character — a space followed by an underscore.) The next statement cancels the Excel cut-copy mode indicator. The last statement simply signals the end of the macro subroutine.



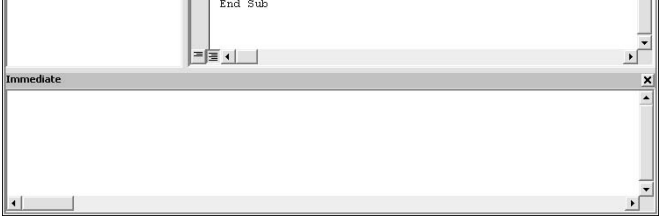


Figure 2-5: The VBE displays the VBA code in Module1 of Book1.

Hey, I didn't record that!

I've noted that the macro recorder is like recording sound on a tape recorder. When you play back an audio tape and listen to you own voice, you invariably say "I don't sound like that." And when you look at your recorded macro, you may see some actions that you didn't think you recorded.

When you recorded the ConvertFormulas example, you didn't change the Operation, SkipBlanks, or Transpose options in the Paste Special dialog box, yet the recorder still recorded them. Don't worry, it happens all the time. When you record an action that includes a dialog box, Excel records all of the options in the dialog box, not just the ones that you change. In later chapters, you'll learn how to remove the extra stuff from a recorded macro.

Modifying the Macro

The macro you've created is fairly useful, saving you a few seconds every time you need to convert formulas to values — but it's also dangerous. After you execute this macro, you may notice that you can't choose the Edit➪Undo command. In other words, if you execute this macro accidentally, you have no way to convert the values back to the original formulas. (Actually, you can develop macros that can be undone with the Edit➪Undo command. That, however, is a bit beyond the scope of this book.)

In this section, you make a minor addition to the macro to prompt users to verify their intentions before the formula-to-value conversion takes place. Issuing such a warning isn't completely foolproof, but it's better than nothing.

You need to provide a pop-up message asking the user to confirm the macro by clicking Yes or No. Fortunately, a VBA statement exists that lets you do this quite easily.

Working in a VBA module is much like working in a word-processing document (except there's no word wrap). You can press Enter to start a new line and the familiar editing keys work as expected.

Here's how you modify the macro to include the warning:

1. **In the VBE, activate Module1.**
2. **Place the cursor at the beginning of the `Selection.Copy` statement.**
3. **Press Enter to insert a new line and then type the following VBA statements:**

```
Answer = MsgBox("Convert formulas to values?", vbYesNo)
If Answer <> vbYes Then Exit Sub
```

To make the new statements line up with the existing statements, press Tab before typing the new statements. Indenting text is optional but it makes your macros easier to read. Your macro should now look like the example in Figure 2-6.

These new statements cause Excel to display a message box with two buttons: Yes and No. The user's button click is stored in a variable named Answer. If the Answer is not equal to Yes, Excel exits the subroutine with no further action (<> represents *not equal to*). Figure 2-7 shows this message box in action.

Activate a worksheet and try out the revised macro to see how it works. To test your macro, you may need to add some more formulas to your worksheet.

Just as you can press Alt+F11 to display the VBE, you can again press Alt+F11 to switch back to Excel.

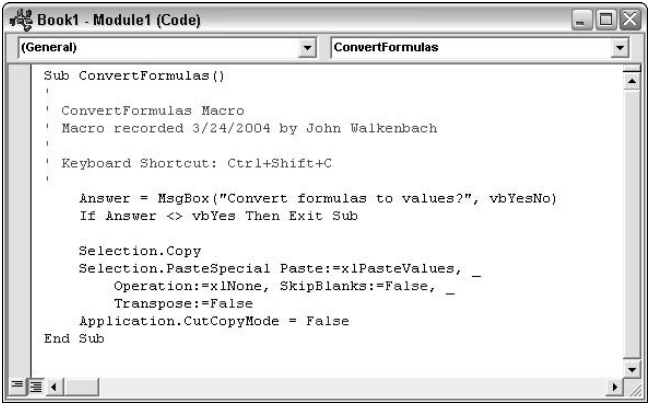
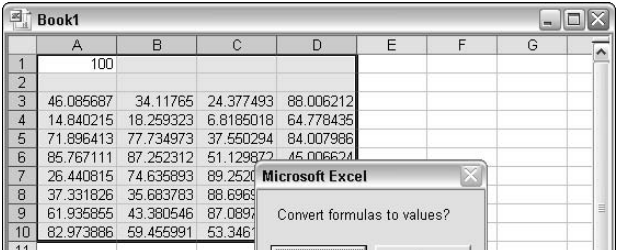


Figure 2-6: The Convert-Formulas macro looks like this after you add to the statements.



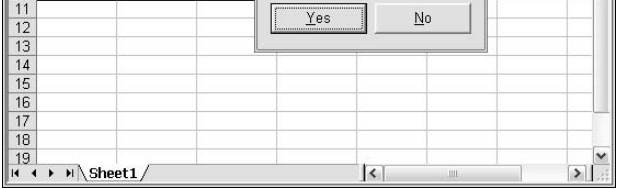


Figure 2-7: The macro displays this message box.

You'll find that clicking the No button cancels the macro, and that the formulas in the selection remain intact. When you click the No button, Excel executes the *Exit Sub* part of the statement. If you click Yes, the macro continues its normal course of action.

If you find this macro useful, save the workbook file.

More about the ConvertFormulas Macro

By the time you finish this book, you'll completely understand how the ConvertFormulas macro works — and you'll be able to develop more-sophisticated macros. For now, I wrap up the example with a few additional points about the macro:

- For this macro to work, its workbook must be open. If you close the workbook, the macro doesn't work (and the Ctrl+Shift+C shortcut has no effect).
- As long as the workbook containing the macro is open, you can run the macro while any workbook is active. In other words, the macro's own workbook doesn't have to be active.
- The macro isn't perfect. One of its flaws is that it generates an error if the selection isn't a range. For instance, if you select a chart and then press Ctrl+Shift+C, the macro grinds to a halt and you see the error message shown in Figure 2-8. In Chapter 14, I show you how to correct this type of problem.

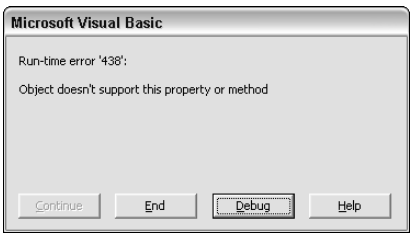


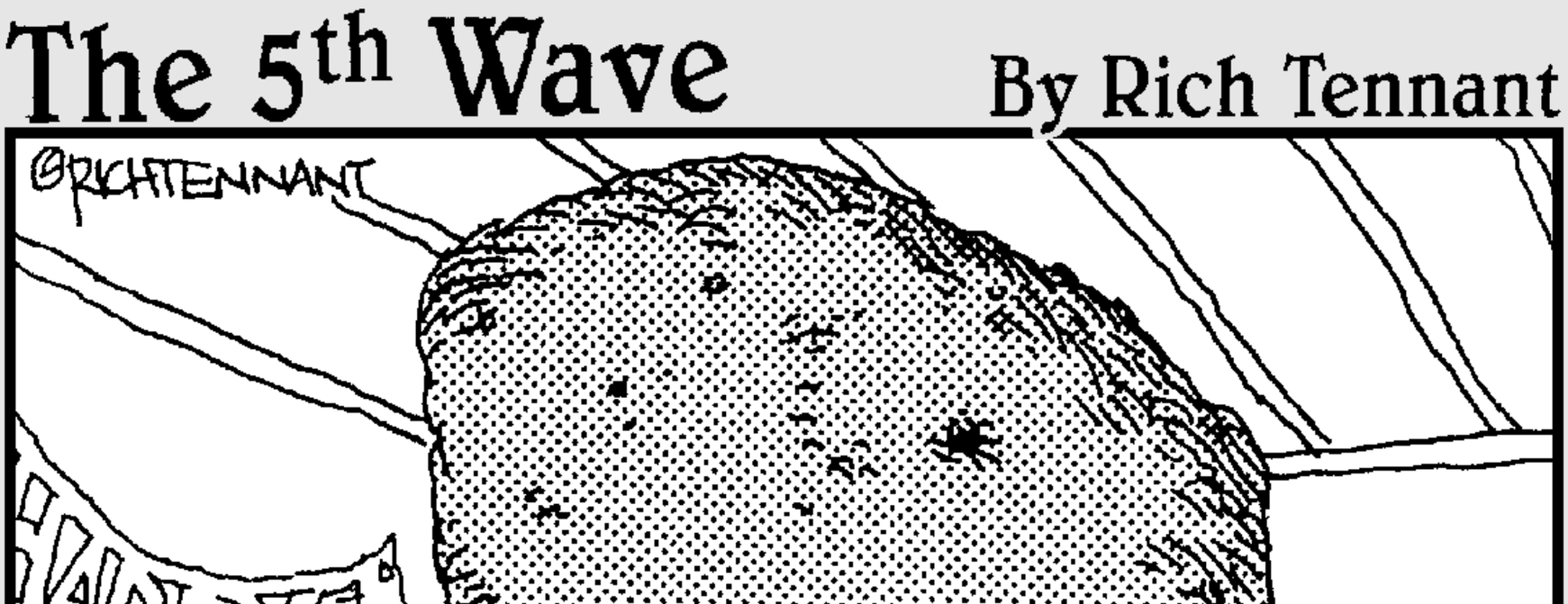
Figure 2-8: The error message is VBA's way of telling you something's wrong.

- Before you started recording the macro, you assigned it a new shortcut key. This is just one of several ways to execute the macro.
- You could enter this macro manually rather than record it. To do so, you need a good understanding of VBA. (Be patient, you'll get there.)
- The two statements you added after the fact are examples of VBA statements that you *cannot* record.
- You could store this macro in your Personal Macro Workbook. If you were to do so, the macro would be available automatically whenever you start Excel. See Chapter 6 for details about your Personal Macro Workbook.
- You could also convert the workbook to an add-in file. (More about this in Chapter 22.)

You've been initiated into the world of Excel programming. (Sorry, there's no secret handshake or decoder ring.) I hope this chapter helps you realize that Excel programming is something you can actually do — and even live to tell about it. Keep reading. Subsequent chapters almost certainly answer any questions you have, and you'll soon understand exactly what you did in this hands-on session.

Part II

How VBA Works with Excel



"I hear some of the engineers from the maintenance dept. project manager baking a birthday fruitcake."

In this part . . .

The next four chapters provide the necessary foundation for discovering the ins and outs of VBA. You find out about modules (the sheets that store your VBA code) and are introduced to the Excel object model (something you won't want to miss). You also discover the difference between subroutines and functions, and you get a crash course in the Excel macro recorder.

Chapter 3

Introducing the Visual Basic Editor

In This Chapter

- Understanding the Visual Basic Editor
- Discovering the Visual Basic Editor parts
- Knowing what goes into a VBA module
- Understanding three ways to get VBA code into a module
- Customizing the VBA environment

As an experienced Excel user, you know a good deal about workbooks, formulas, charts, and other Excel goodies. Now it's time to expand your horizons and explore an entirely new aspect of Excel: the Visual Basic Editor (VBE). In this chapter, you find out how to work with the VBE, and get down to the nitty-gritty of entering VBA code.

What Is the Visual Basic Editor?

The Visual Basic Editor is a separate application where you write and edit your Visual Basic macros. It works seamlessly with Excel. By *seamlessly,* I mean that Excel takes care of opening VBE when you need it.

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 menus within Excel. To do this, choose Tools➪Macro➪Visual Basic Editor.

Understanding VBE components

Figure 3-1 shows the VBE program window, with some of the key parts identified. Because so much is going on in the VBE, maximize the window to see as much as possible.

Chances are your VBE program window won't look exactly like the window shown in Figure 3-1. This window is highly customizable — you can hide, resize, dock, rearrange, and so on in the window.

Actually, the VBE has even more parts than are shown in Figure 3-1. I discuss these additional components in both Chapter 13 and in Part IV.

Menu bar

The VBE menu bar, of course, works 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 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.

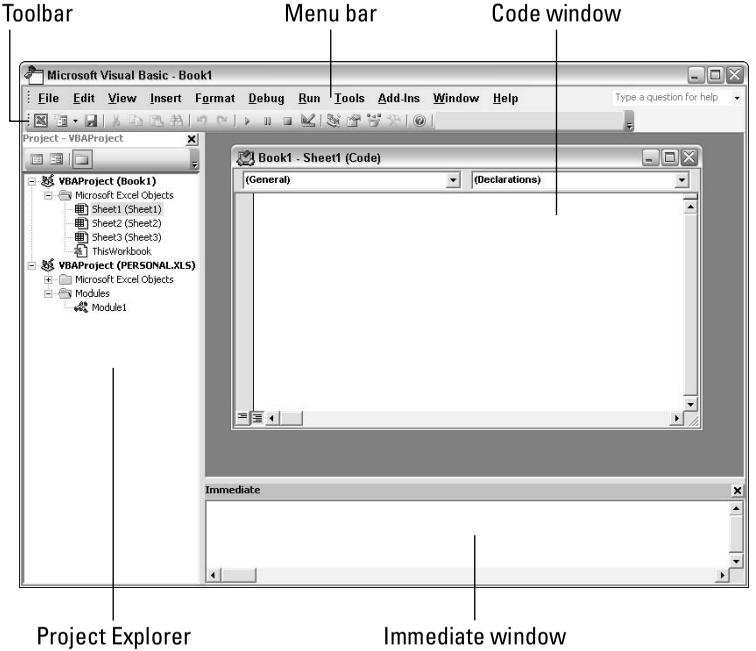


Figure 3-1: The VBE window is your customizable friend.

Toolbar

The Standard toolbar, which is directly under the menu bar by default (see Figure 3-1), is one of four VBE toolbars available. VBE toolbars work just like those in Excel: You can customize them, move them around, display other toolbars, and so on. Use the View➪Toolbars command to work with VBE toolbars.

Project Explorer window

The Project Explorer window displays a tree diagram that consists of every workbook currently open in Excel (including add-ins and hidden workbooks). I discuss this window in more detail in "Working with the Project Explorer," later in this chapter.

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

Code window

A Code window (sometimes known as a Module 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 Explorer window. For example, to view the Code window for the Sheet1 object, double-click Sheet1 in the Project Explorer window. Unless you've added some VBA code, the Code window will be empty.

You find out more about Code windows later in this chapter'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 get it out of the way.

In Chapter 13, I discuss the Immediate window in detail. It may just become your good friend!

Working with the Project Explorer

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 Explorer window. Contract a project by clicking the minus sign (-) to the left of a project's name. Figure 3-2 shows a Project Explorer window with three projects listed.

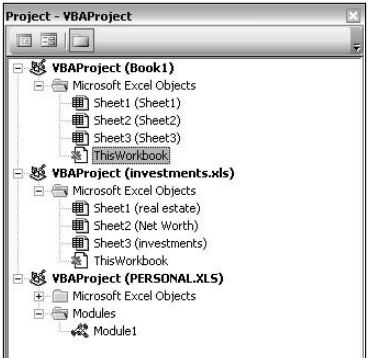


Figure 3-2: This Project Explorer window lists three projects — Book1, investments.xls, and PERSONAL.XLS.