LEARNING MADE EASY



3rd Edition

Excel® Macros

for dummies®

A Wiley Brand



Build and customize macros to make Excel fly

Learn to write macros and become the office expert

Automate repetitive tasks to save time and errors

Dick Kusleika



Excel® Macros

3rd Edition

by Dick Kusleika

for dummies®
A Wiley Brand

Excel® Macros For Dummies®, 3rd Edition

Published by: **John Wiley & Sons, Inc.,** 111 River Street, Hoboken, NJ 07030-5774, www.wiley.com

Copyright © 2022 by John Wiley & Sons, Inc., Hoboken, New Jersey

Published simultaneously in Canada

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without the prior written permission of the Publisher. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at http://www.wiley.com/go/permissions.

Trademarks: Wiley, For Dummies, the Dummies Man logo, Dummies.com, Making Everything Easier, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and may not be used without written permission. Microsoft and Excel are registered trademarks of Microsoft Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc. is not associated with any product or vendor mentioned in this book.

LIMIT OF LIABILITY/DISCLAIMER OF WARRANTY: WHILE THE PUBLISHER AND AUTHORS HAVE USED THEIR BEST EFFORTS IN PREPARING THIS WORK, THEY MAKE NO REPRESENTATIONS OR WARRANTIES WITH RESPECT TO THE ACCURACY OR COMPLETENESS OF THE CONTENTS OF THIS WORK AND SPECIFICALLY DISCLAIM ALL WARRANTIES, INCLUDING WITHOUT LIMITATION ANY IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. NO WARRANTY MAY BE CREATED OR EXTENDED BY SALES REPRESENTATIVES, WRITTEN SALES MATERIALS OR PROMOTIONAL STATEMENTS FOR THIS WORK. THE FACT THAT AN ORGANIZATION, WEBSITE, OR PRODUCT IS REFERRED TO IN THIS WORK AS A CITATION AND/OR POTENTIAL SOURCE OF FURTHER INFORMATION DOES NOT MEAN THAT THE PUBLISHER AND AUTHORS ENDORSE THE INFORMATION OR SERVICES THE ORGANIZATION, WEBSITE, OR PRODUCT MAY PROVIDE OR RECOMMENDATIONS IT MAY MAKE. THIS WORK IS SOLD WITH THE UNDERSTANDING THAT THE PUBLISHER IS NOT ENGAGED IN RENDERING PROFESSIONAL SERVICES. THE ADVICE AND STRATEGIES CONTAINED HEREIN MAY NOT BE SUITABLE FOR YOUR SITUATION. YOU SHOULD CONSULT WITH A SPECIALIST WHERE APPROPRIATE. FURTHER, READERS SHOULD BE AWARE THAT WEBSITES LISTED IN THIS WORK MAY HAVE CHANGED OR DISAPPEARED BETWEEN WHEN THIS WORK WAS WRITTEN AND WHEN IT IS READ. NEITHER THE PUBLISHER NOR AUTHORS SHALL BE LIABLE FOR ANY LOSS OF PROFIT OR ANY OTHER COMMERCIAL DAMAGES, INCLUDING BUT NOT LIMITED TO SPECIAL, INCIDENTAL, CONSEQUENTIAL, OR OTHER DAMAGES.

For general information on our other products and services, please contact our Customer Care Department within the U.S. at 877-762-2974, outside the U.S. at 317-572-3993, or fax 317-572-4002. For technical support, please visit https://hub.wiley.com/community/support/dummies.

Wiley publishes in a variety of print and electronic formats and by print-on-demand. Some material included with standard print versions of this book may not be included in e-books or in print-on-demand. If this book refers to media such as a CD or DVD that is not included in the version you purchased, you may download this material at http://booksupport.wiley.com. For more information about Wiley products, visit www.wiley.com.

Library of Congress Control Number: 2022931276

ISBN: 978-1-119-84443-3 (pbk); ISBN: 978-1-119-84444-0 (ePDF); ISBN: 978-1-119-84445-7 (ePub)

Excel® Macros For Dummies®

To view this book's Cheat Sheet, simply go to www.dummies.com and search for "Excel Macros For Dummies Cheat Sheet" in the Search box.

Table of Contents

- Cover
- Title Page
- Copyright
- Introduction
  - About This Book
  - Foolish Assumptions
  - Icons Used in This Book
  - Beyond the Book
  - Where to Go from Here
- Part 1: Holy Macro Batman!
  - Chapter 1: Macro Fundamentals
    - Choosing to Use a Macro
    - Macro Recording Basics
    - Comparing Absolute and Relative Macro Recording
    - Understanding Macro Security
    - Storing and Running Macros
    - Exploring Macro Examples
  - Chapter 2: Getting Cozy with the Visual Basic Editor
    - Working in the Visual Basic Editor
    - Working with the Project Explorer
    - Working with a Code Pane
    - Customizing the VBE
  - Chapter 3: The Anatomy of Macros
    - A Brief Overview of the Excel Object Model
    - A Brief Look at Variables
    - Understanding Event Procedures
    - Error Handling in a Nutshell
- Part 2: Making Short Work of Workbook Tasks

Chapter 4: Working with Workbooks

- Installing Macros
- Creating a New Workbook from Scratch
- Saving a Workbook when a Particular Cell Is Changed
- Saving a Workbook before Closing
- Protecting a Worksheet on Workbook Close
- Unprotecting a Worksheet
- Opening a Workbook to a Specific Tab
- Opening a Specific Workbook Defined by the User
- Determining Whether a Workbook Is Already Open
- Determining Whether a Workbook Exists in a Directory
- Closing All Workbooks at Once
- Printing All Workbooks in a Directory
- Preventing the Workbook from Closing Until a Cell Is Populated
- Creating a Backup of the Current Workbook with Today's Date

Chapter 5: Working with Worksheets

- Installing Macros
- Adding and Naming a New Worksheet
- Deleting All but the Active Worksheet
- Hiding All but the Active Worksheet
- Unhiding All Worksheets in a Workbook
- Moving Worksheets Around
- Sorting Worksheets by Name
- Grouping Worksheets by Color
- Copying a Worksheet to a New Workbook
- Creating a New Workbook for Each Worksheet
- Printing Specified Worksheets
- Protecting All Worksheets
- Unprotecting All Worksheets
- Creating a Table of Contents for Your Worksheets
- Zooming In and Out of a Worksheet with Double-Click
- Highlighting the Active Row and Column

Part 3: One-Touch Data Manipulation

Chapter 6: Feeling at Home on the Range

- Installing Macros
- Selecting and Formatting a Range
- Creating and Selecting Named Ranges
- Looping Through a Range of Cells
- Inserting Blank Rows in a Range
- Unhiding All Rows and Columns

Deleting Blank Rows
Deleting Blank Columns
Limiting Range Movement to a Particular Area
Selecting and Formatting All Formulas in a Workbook
Finding and Selecting the First Blank Row or Column

Chapter 7: Manipulating Data with Macros

Installing Macros
Copying and Pasting a Range
Converting All Formulas in a Range to Values
Text to Columns on All Columns
Converting Trailing Minus Signs
Trimming Spaces from All Cells in a Range
Truncating Zip Codes to the Left Five
Padding Cells with Zeros
Replacing Blanks Cells with a Value
Appending Text to the Left or Right of Your Cells
Cleaning Up Non-Printing Characters
Highlighting Duplicates in a Range of Data
Hiding All but Rows Containing Duplicate Data
Selectively Hiding AutoFilter Drop-down Arrows
Copying Filtered Rows to a New Workbook
Showing Filtered Columns in the Status Bar

Part 4: Macro-Charging Reports and Emails

Chapter 8: Automating Common Reporting Tasks

Installing Macros
Refreshing All PivotTables in a Workbook
Creating a List of PivotTables
Adjusting All Pivot Data Field Titles
Setting All Data Items to Sum
Applying Number Formatting for All Data Items
Sorting All Fields in Alphabetical Order
Applying a Custom Sort to Data Items
Applying PivotTable Restrictions
Applying Pivot Field Restrictions
Automatically Deleting PivotTable Drill-Down Sheets
Printing a PivotTable for Each Report Filter Item
Creating a New Workbook for Each Report Filter Item
Resizing All Charts on a Worksheet
Aligning a Chart to a Specific Range
Creating a Set of Disconnected Charts
Printing All Charts on a Worksheet

Chapter 9: Sending Emails from Excel

- Installing Macros
- Mailing the Active Workbook as an Attachment
- Mailing a Specific Range as an Attachment
- Mailing a Single Sheet as an Attachment
- Sending Mail with a Link to Your Workbook
- Mailing All Email Addresses in Your Contact List
- Saving All Attachments to a Folder
- Saving Certain Attachments to a Folder

Chapter 10: Wrangling External Data with Macros

- Working with External Data Connections
- Using Macros to Create Dynamic Connections
- Iterating through All Connections in a Workbook
- Using ADO and VBA to Pull External Data
- Working with Text Files

Part 5: The Part of Tens

Chapter 11: Ten Handy Visual Basic Editor Tips

- Applying Block Comments
- Copying Multiple Lines of Code at Once
- Jumping between Modules and Procedures
- Teleporting to Your Functions
- Staying in the Right Procedure
- Stepping through Your Code
- Stepping to a Specific Line in Your Code
- Stopping Your Code at a Predefined Point
- Seeing the Beginning and End of Variable Values
- Turning Off Auto Syntax Check

Chapter 12: Ten Places to Turn for Macro Help

- Let Excel Write the Macro for You
- Use the VBA Help Files
- Pilfer Code from the Internet
- Leverage User Forums
- Visit Expert Blogs
- Mine YouTube for Video Training
- Attend Live and Online Training Classes
- Learn from the Microsoft Office Dev Center
- Dissect the Other Excel Files in your Organization
- Ask Your Local Excel Genius

Chapter 13: Ten Ways to Speed Up Your Macros

- Halt Calculations

Disable Screen Updating
Turn Off Status Bar Updates
Tell Excel to Ignore Events
Hide Page Breaks
Suspend PivotTable Updates
Steer Clear of Copy and Paste
Use the With Statement
Don’t Explicitly Select Objects
Avoid Excessive Trips to the Worksheet

Index

About the Author

Advertisement Page

Connect with Dummies

End User License Agreement

List of Illustrations

Chapter 1

FIGURE 1-1: The Record Macro dialog box.
FIGURE 1-2: Your pre-totaled worksheet containing two tables.
FIGURE 1-3: Your post-totaled worksheet.
FIGURE 1-4: The Excel Macro dialog box.
FIGURE 1-5: Recording a macro with relative references.
FIGURE 1-6: The Trusted Locations window allows you to add directories that are...
FIGURE 1-7: You can find the form controls on the Developer tab.
FIGURE 1-8: Assign a macro to the newly added button.
FIGURE 1-9: Adding a macro to the Quick Access Toolbar.
FIGURE 1-10: Use macros to build buttons that help users navigate your reports.
FIGURE 1-11: This report allows users to choose their perspective.
FIGURE 1-12: The macros behind these buttons rearrange the data fields in a Piv...
FIGURE 1-13: Offering prerecorded views not only saves time and effort, but it ...
FIGURE 1-14: You can give your audience a choice in how they view data.

Chapter 2

FIGURE 2-1: The VBE with significant elements identified.
FIGURE 2-2: The Project Explorer with two projects open, expanded to show their...
FIGURE 2-3: Code modules are visible in the Project Explorer in a folder called...
FIGURE 2-4: The Editor tab in the Options dialog box.
FIGURE 2-5: Change the way VBE’s looks with the Editor Format tab.
FIGURE 2-6: The General tab of the Options dialog box.

FIGURE 2-7: The Docking tab of the Options dialog box.

Chapter 3

FIGURE 3-1: Getting to the built-in module for a worksheet.
FIGURE 3-2: The default SelectionChange event for the Worksheet object.
FIGURE 3-3: Click the Event drop-down list to choose the most appropriate event...
FIGURE 3-4: The default Open event for the Workbook object.
FIGURE 3-5: Click the Event drop-down list to choose the most appropriate event...

Chapter 4

FIGURE 4-1: Changing any cell in range C5:C16 forces the workbook to save.
FIGURE 4-2: Enter or paste your code in the Worksheet_Change event Code pane.
FIGURE 4-3: A message box opens when you attempt to close the workbook.
FIGURE 4-4: Type or paste your code in the Workbook_BeforeClose event Code pane...
FIGURE 4-5: Type or paste your code in the Workbook BeforeClose event Code.
FIGURE 4-6: Type or paste your code in the Workbook Open event Code pane.
FIGURE 4-7: You want your workbook to automatically open to the sheet called St...
FIGURE 4-8: Type or paste your code in the Workbook Open event Code pane.
FIGURE 4-9: The macro opens the Open dialog box.
FIGURE 4-10: You can avoid this annoying message box when opening a workbook th...
FIGURE 4-11: You can prevent your workbook from closing until a specific cell i...
FIGURE 4-12: Type or paste your code in the Workbook_BeforeClose event Code pan...

Chapter 5

FIGURE 5-1: Omit the Application.DisplayAlerts = False line in the macro to ens...
FIGURE 5-2: Without a macro, you’re stuck using Excel’s Unhide dialog box to un...
FIGURE 5-3: It’s often useful to have your worksheets sorted in alphabetical or...
FIGURE 5-4: You can right-click any worksheet to choose a tab color for the she...
FIGURE 5-5: The Protect Sheet command is disabled if you try to protect more th...
FIGURE 5-6: The Unprotect Sheet command is disabled if you try to unprotect mor...
FIGURE 5-7: A table of contents can help you more easily navigate your workbook...
FIGURE 5-8: Type or paste your code into the Worksheet_BeforeDoubleClick event ...
FIGURE 5-9: A highlighted row and column makes it easy to track data horizontal...

Chapter 6

FIGURE 6-1: Click the Define Name command to name a chosen range.
FIGURE 6-2: Give your range a name.
FIGURE 6-3: The Name Manager dialog box lists all the applied named ranges.
FIGURE 6-4: Type or paste your code in the Workbook Open event Code pane.
FIGURE 6-5: The Go To Special dialog box.
FIGURE 6-6: Choosing Formulas in the Go To Special dialog box tells Excel to se...
FIGURE 6-7: You can use a macro to tell Excel to dynamically find the first ava...

Chapter 7

FIGURE 7-1: Imported numbers are sometimes formatted as text.

FIGURE 7-2: Click the Text to Columns command.
FIGURE 7-3: Clicking Finish in the Text to Columns dialog box corrects incorrec...
FIGURE 7-4: This macro dynamically finds and highlights the duplicate values in...
FIGURE 7-5: This macro ensures that only those rows that contain duplicate valu...
FIGURE 7-6: The standard AutoFilter functionality adds drop-down arrows to all ...
FIGURE 7-7: With a little VBA, you can choose to hide certain AutoFilter drop-d...
FIGURE 7-8: This macro lists all filtered columns in the status bar.
FIGURE 7-9: Type or paste your code in the Worksheet_Calculate event Code pane.

Chapter 8

FIGURE 8-1: A PivotTable inventory summary.
FIGURE 8-2: Excel shows the enumeration choices.
FIGURE 8-3: Type or paste your code in the Worksheet_BeforeDoubleClick event Co...
FIGURE 8-4: Enter or paste your code in the Workbook_BeforeClose event Code pan...
FIGURE 8-5: The Selection pane allows you to see all of your chart objects' nam...

Chapter 10

FIGURE 10-1: Show legacy wizards in the Ribbon.
FIGURE 10-2: Choose the source database that contains the data you want importe...
FIGURE 10-3: Select the Access object you want to import.
FIGURE 10-4: Choosing how and where to view your Access data.
FIGURE 10-5: Data imported from Access.
FIGURE 10-6: As long as a connection to your database is available, you can upd...
FIGURE 10-7: Select the SQL command type and enter your SQL statement.
FIGURE 10-8: Designate a cell that will trap the criteria selection.
FIGURE 10-9: You now have an easy-to-use mechanism to pull external data for a ...
FIGURE 10-10: Select the latest version of the Microsoft ActiveX Data Objects L...

Chapter 11

FIGURE 11-1: A single apostrophe in front of any line turns that line into a co...
FIGURE 11-2: The Edit toolbar allows you to select entire blocks of code, and t...
FIGURE 11-3: Holding down the Ctrl key while dragging code creates a copy of th...
FIGURE 11-4: Pressing Shift+F2 with your cursor on a function or variable name ...
FIGURE 11-5: Use the Procedure View button to limit scrolling to just the activ...
FIGURE 11-6: Press F8 key to step through each line of your macro at your own p...
FIGURE 11-7: You can click and drag the yellow arrow while stepping through you...
FIGURE 11-8: A breakpoint is marked by a dot in the left margin along with shad...
FIGURE 11-9: Showing the beginning and ending characters in a variable tooltip.
FIGURE 11-10: Leaving a line of code unfinished, even for a second, results in ...
FIGURE 11-11: Uncheck the Auto Syntax Check option to prevent warning messages ...

Introduction

In its broadest sense, a *macro* is a sequence of instructions that automates some aspect of Excel so that you can work more efficiently and with fewer errors. You might create a macro, for example, to format and print a month-end sales report. After you develop the macro, you can execute it to perform many time-consuming procedures automatically.

Macros are written in VBA, which stands for Visual Basic for Applications. VBA is a programming language developed by Microsoft and a tool used to develop programs that control Excel.

Excel programming terminology can be a bit confusing. For example, VBA is a programming language but also serves as a macro language. What do you call something written in VBA and executed in Excel? Is it a macro or is it a program? Excel's Help system often refers to VBA procedures as *macros*, so this is the terminology used in this book.

You also see the term *automate* throughout this book. This word means that a series of steps are completed automatically. For example, if you write a macro that adds color to some cells, prints the worksheet, and then removes the color, you have automated those three steps.

You're probably aware that people use Excel for thousands of different tasks. Here are just a few examples:

- Keeping lists of things, such as customer names and transactions
- Budgeting and forecasting
- Analyzing scientific data
- Creating invoices and other forms
- Developing charts from data

The list could go on and on. The point is simply that Excel is used for a wide variety of tasks, and everyone reading this book has different needs and expectations regarding Excel. One thing most readers have in common, however, is the need to automate some aspect of Excel, which is what macros (and this book) are all about.

About This Book

This book approaches the topic of Excel macros with the recognition that programming VBA takes time and practice — time that you may not have right now. In fact, many analysts don't have the luxury of taking a few weeks to become expert at VBA. So instead of the same general overview of VBA topics, this book provides some of the most commonly used real-world Excel macros.

Each section in the book outlines a common problem and provides an Excel macro to solve the problem — along with a detailed explanation of how the macro works and where to use it.

Each section presents the following:

- The problem
- The macro solution
- How the macro works

After reading each section, you'll be able to

- Immediately implement the required Excel macro
- Understand how the macro works
- Reuse the macro in other workbooks or with other macros

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 to learn 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.

Foolish Assumptions

I make three assumptions about you as the reader:

- You've installed Microsoft Excel.
- You have some familiarity with the basic concepts of data analysis, such as working with tables, aggregating data, creating formulas, referencing cells, filtering, and sorting.
- You have an Internet connection so you can download the sample files.

Icons Used in This Book



Tip icons cover tricks or techniques related to the current discussion.



Remember icons indicate notes or asides that are important to keep in mind.



Warning icons hold critical information about pitfalls you will want to avoid.



The icons marks particularly geeky parts of Excel and VBA. If you’re into the technical aspects, read on. If you’re not, you can completely skip paragraphs marked

with this paragraph and still get your macros to work.

Beyond the Book

In addition to the material in the print or e-book you're reading, this product comes with more online goodies:

- **Sample files:** Each macro in this book has an associated sample file that enables you to see the macro working and to review the code. You can use the sample files also to copy and paste the code into your environment (as opposed to typing each macro from scratch). Download the sample files at:
  www.dummies.com/go/excelmacrosfd3e

  Each macro in this book has detailed instructions on where to copy and paste the code. In general terms, you open the sample file associated with the macro, open the Visual Basic Editor (by pressing Alt+F11), and copy the code. Then open your workbook, open the Visual Basic Editor, and paste the code in the appropriate location.

  REMEMBER Note that in some macros, you need to change the macro to suit your situation. For example, in the macro that prints all workbooks in a directory (see Chapter 4), you point to the C:\Temp\ directory. Before using this macro, you must edit it to point to your target directory.

  TIP If a macro is not working for you, most likely a component of the macro needs to be changed. Pay special attention to range addresses, directory names, and any other hard-coded names.
- **Cheat sheet:** The cheat sheet offers shortcut keys that can help you work more efficiently in Excel's Visual Basic Editor. You can find the cheat sheet by visiting www.dummies.com and searching for "Excel Macros Cheat Sheet."

Where to Go from Here

If you're completely new to Excel macros, start with Part 1 (Chapters 1–3) to get the fundamentals you need to understand the macros in this book. There, you gain a concise understanding of how macros and VBA work, along with the basic foundation you need to implement the macros provided in this book.

If you've got some macro experience and want to dive right into the macro examples, feel free to peruse Chapters 4–9 and search for the task or macro that looks interesting to you. Don't worry. Each macro example stands on its own within its own section that gives you all the guidance you need to understand and implement the code in your own workbook.

Visit Part 2 if you're interested in macros that automate common workbook and worksheet

tasks to save time and gain efficiencies.

Explore Part 3 to find macros that navigate ranges, format cells, and manipulate the data in your workbooks.

If you want to find macros that work with PivotTables, charts, and emails, thumb through the macros in Part 4 where you discover macros that automate redundant PivotTable and chart tasks, as well as macros that send emails and connect to external data sources.

Don't forget to hit Part 5 for some useful tips and advice on how to get the most out of your new macro skills.

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

- **Any file that contains a macro must have the .xlsm file extension.** See the section on macro-enabled file extensions in Chapter 1 for more information.
- **Excel does not run macros until they are enabled.** As you implement these macros, you and your customers must comply with Excel's macro security measures. See the section in Chapter 1 on macro security in Excel for details.
- **You cannot undo macro actions.** When working in Excel, you can often undo the actions you've taken because Excel keeps a log (called the undo stack) recording your last 100 actions. However, running a macro automatically destroys the undo stack, so you can't undo the actions you take in a macro.
- **You need to tweak the macros to fit your workbook.** Many of the macros reference example sheet names and ranges that you may not have in your workbook. Be sure to replace references such as "Sheet 1" or Range ("A1") with the sheet names and cell addresses you are working with in your own workbooks.

Part 1

Holy Macro Batman!

IN THIS PART …

Build a foundation for your macro skills with fundamental macro recording concepts.

Get a solid understanding of the ground rules for using and distributing macros in Excel.

Explore Excel’s coding environment with a deep-dive of the Visual Basic Editor.

Explore how to leverage the Excel object model to start writing your own macros from scratch.

Understand the roles played by variables, events, and error handling in macro development.

Chapter 1
Macro Fundamentals

IN THIS CHAPTER

- **Choosing macros**
- **Recording macros**
- **Understanding macro security**
- **Finding out where to store and how to run macros**
- **Exploring macro examples**

A *macro* is essentially a set of instructions or code that you create to tell Excel to execute any number of actions. In Excel, macros can be written or recorded. The key word here is recorded.

Recording a macro is like programming a phone number into your smartphone. You first manually dial and save a number. Then when you want, you can redial those numbers with the touch of a button. With macro recording, 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 code (also known as Visual Basic for Applications or VBA). After a macro is recorded, you can play back those actions anytime you want.

In this chapter, you explore macros and find out how you can use macros to automate your recurring processes to simplify your life.

Choosing to Use a Macro

The first step in using macros is admitting you have a problem. Actually, you may have several problems:

- **Problem 1— repetitive tasks:** As each new month rolls around, you have to make the donuts (that is, crank out those reports). You have to import that data. You have to update those PivotTables. You have to delete those columns, and so on. With a macro you could have those more redundant parts of your monthly process processes done automatically.
- **Problem 2 — you're making mistakes:** When you go hand-to-hand combat with Excel, you're bound to make mistakes. When you're repeatedly applying formulas, sorting, and moving things around manually, there's always that risk of catastrophe. Add to that the looming deadlines and constant change requests, and your error rate goes up. Or you could calmly record a macro, ensure that everything is running correctly, and then forget it. The macro performs every action the same way every time you run it, reducing the chance of errors.

- **Problem 3 — awkward navigation:** You often create reports for an audience that probably has a limited knowledge of Excel. It's always helpful to make your reports more user-friendly. Macros can be used to dynamically format and print worksheets, navigate to specific sheets in your workbook, or even save the open document in a specified location. Your audience will appreciate these little touches that help make perusal of your workbooks a bit more pleasant.

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 File ⇒ Options.**
2. **In the Excel Options dialog box, click 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.

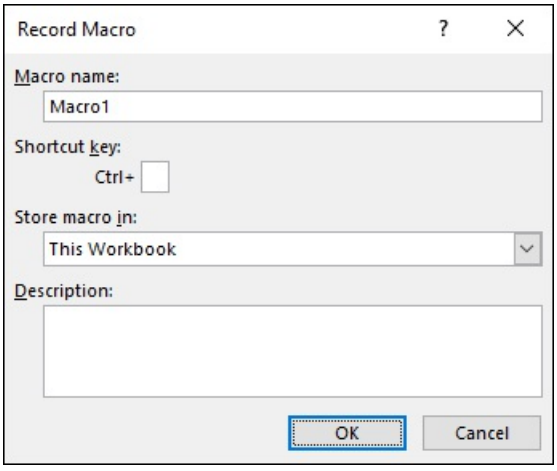


FIGURE 1-1: The Record Macro dialog box.

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

- **Macro Name:** 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.



REMEMBER You have to follow a few rules when naming a macro. The first character must be a letter. Generally, special characters other than underscore aren't allowed. And the total number of characters can't be more than 255, although hopefully you don't get close to that limit.

- **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 if you use this field, 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 chapter). You can also choose New Workbook to tell Excel to create a new workbook to store the macro or Personal Macro Workbook, a special workbook used to store macros you want access to all the time. See "Storing and Running Macros" later in this chapter for more on the Personal Macro Workbook.
- **Description:** This field is optional, but it can come in handy if you have numerous macros in a workbook 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.**
   You do this by entering uppercase N in the edit box labeled Shortcut Key.
3. **Click OK.**
   This closes the Record Macro dialog box and begins recording your actions.
4. **Select cell B3 on your worksheet, type your name into the selected cell, and then press Enter.**
5. **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. (See Chapter 2 to find out more about the Visual Basic Editor.) You can activate the VB Editor in one of three ways:

- Press Alt+F11.
- Choose Developer ⇒ Code ⇒ Visual Basic.
- Choose Developer ⇒ Code ⇒ Macros, select a macro, and click Edit.

In the VB Editor, the Project window displays a list of all open workbooks and add-ins. If the Project Explorer isn't visible, choose View ⇒ Project Explorer. 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
'
    Range("B3").Select
    ActiveCell.FormulaR1C1 = "Dick Kusleika"
    Range("B4").Select
End Sub
```

The macro recorded is a Sub procedure 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 three VBA statements, the second of which is:

```
ActiveCell.FormulaR1C1 = "Dick Kusleika"
```

The first statement is selecting cell B3. The last statement is pressing Enter after you enter your name, which moves the active cell down one row. The middle statement, the one that does all the work, causes the name you typed while recording to be inserted into the active cell.

Editing the macro

After you record a macro, you can make changes to it. The macro you recorded in the previous section always inserts your name into cell B3. Edit the macro so that it enters your name in whatever cell you happen to be in when you run it. To do that, delete the first and third lines of the macro. The edited macro appears as follows:

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

This macro inserts text into the active cell because the first Select statement was removed. That same cell remains active because the second Select statement was removed.

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.

Comparing Absolute and Relative Macro Recording

Excel has two modes for recording — absolute reference and relative reference. These modes affect what statements Excel generates when you select cell while recording. In this section, I discuss the two modes and when to use them.

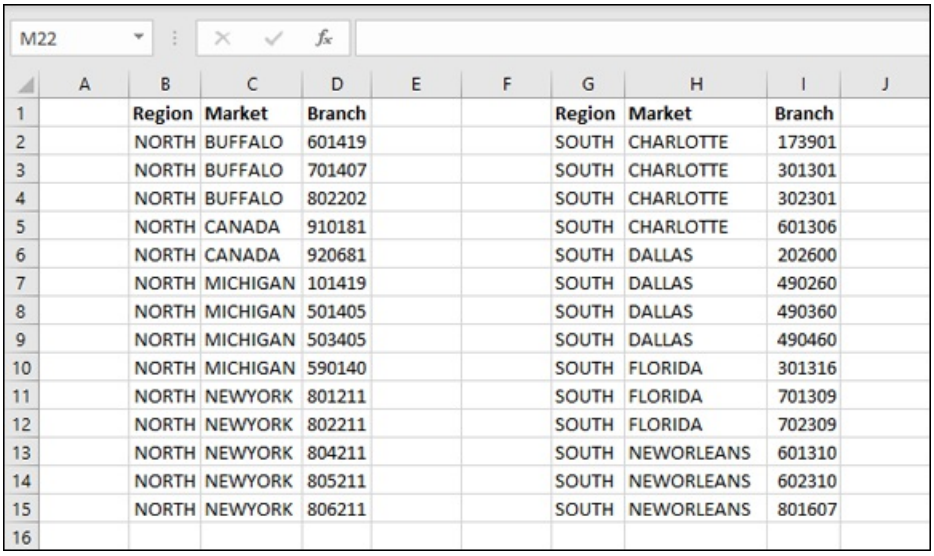
Recording macros with absolute references

In the example in the preceding section, you selected cell B3 while recording a macro and Excel dutifully recorded a statement that selects cell B3. This is an example of an absolute reference and it's the default mode when recording macros. 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. Open the Chapter 1 Sample File.xlsx file and record a macro that counts the rows in the Branchlist worksheet. (See Figure 1-2.)



TIP The sample dataset used in this chapter can be found on the book's companion website.



	A	B	C	D	E	F	G	H	I	J
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										

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

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:

   1. **Select cell A16 and type** Total **in the cell.**
   2. **Select the first empty cell in Column D (D16) and enter** = 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.
6. **Click Stop Recording on the Developer tab to stop recording the macro.**

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

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

1. **Click Macros on 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 (G1:I15 in Figure 1-3). Why? Because you recorded the macro using absolute references.

To understand what this means, examine the underlying code. To examine the code, click Macros on the Developer tab to open the Macro dialog box, as shown in Figure 1-4.

	A	B	C	D	E	F	G	H	I	J
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						
17										

FIGURE 1-3: Your post-totaled worksheet.

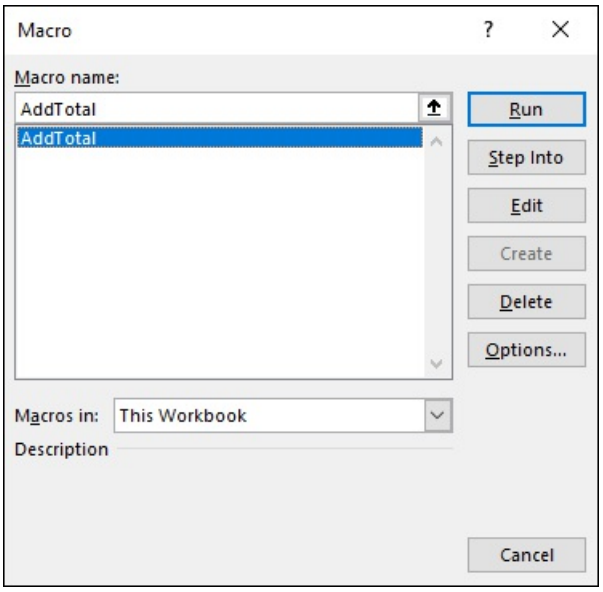


FIGURE 1-4: The Excel Macro dialog box.

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 2 and 4 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. Then, use the following steps to record a relative-reference macro:

1. **Click Use Relative References from the Developer tab, as shown in Figure 1-5.**
2. **Before recording, make sure cell A1 is selected.**
3. **Click Record Macro from the Developer tab.**
4. **Name the macro AddTotalRelative.**
5. **Choose This Workbook in the Macros In drop-down list.**
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.**

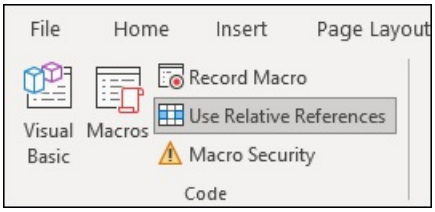


FIGURE 1-5: Recording a macro with relative references.

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

Click 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. Take a look at what the relevant parts of this VBA code really mean.

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). Excel doesn't select a cell with a specific address as it did when recording an absolute reference macro.

Between Offset and Select on the second line is Range("A1"). This is Excel recording that you only selected one cell — the first cell of the range that is offset from the active cell. It's a quirk of the Macro Recorder and isn't necessary when you select only one cell. (The Macro Recorder records a lot of unnecessary code.) If you had selected, say A16:B17 instead of just A16, it would have recorded:

```
ActiveCell.Offset(15, 0).Range("A1:B2").Select
```

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

1. **Select cell A1.**
2. **Click Macros on the Developer tab.**
3. **Select the AddTotalRelative macro.**
4. **Click the Run button.**
5. **Select cell F1.**
6. **Click Macros on the Developer tab.**
7. **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.

Understanding Macro Security

At this point, you should feel comfortable recording your own Excel macros. In the wrong hands, macros can be destructive. Because macros are so powerful, some people have used them to create macro viruses that can be harmful to your computer. For that reason, Microsoft has built some security measures around macros. Here are some important security concepts you need to keep in mind when working with macros.

Macro-enabled file extensions

Excel's default file format, called Excel Workbook, has a .xlsx file extension. 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 removed 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.

Trusted documents

Excel allows you to flag a document as trusted. 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, you see a yellow bar message under the Ribbon stating that macros (active content) have been disabled. If you have the Visual Basic Editor open, you'll get a dialog box instead of the yellow bar.

If you click Enable, the workbook 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 users of your macros. 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 workbook 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. **Click the Macro Security button on the Developer tab.**

The Trust Center dialog box opens.

2. **Click the Trusted Locations option on the left.**

   This opens the Trusted Locations window (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.**
5. **Add an optional description, click OK to close the Microsoft Office Trusted Location dialog, and then click OK again to close the Trust Center dialog.**

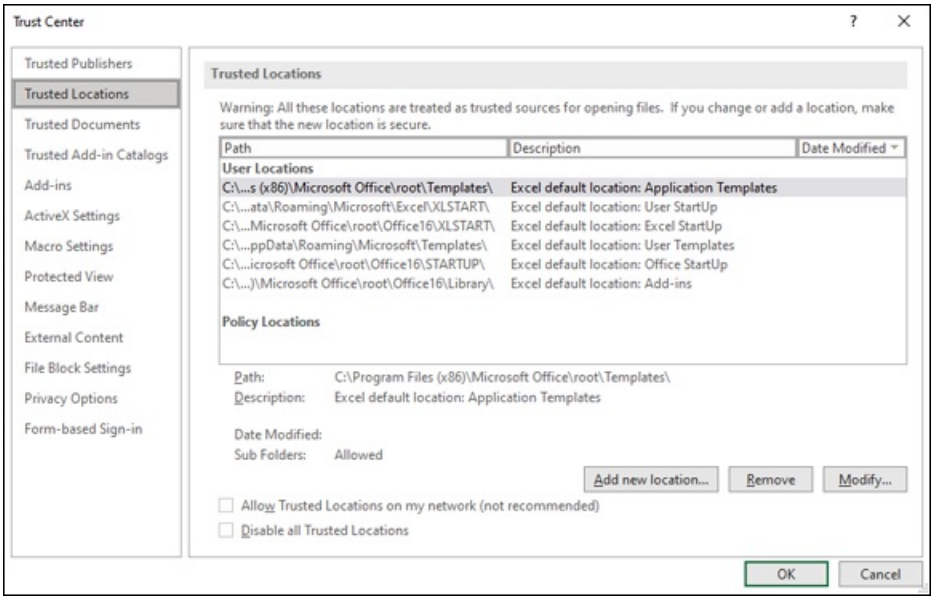


FIGURE 1-6: The Trusted Locations window allows you to add directories that are considered trusted.

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

Storing and Running Macros

Macros are stored in a workbook. Even though you work with macros in the Visual Basic Editor, the code is stored in the same file with all the worksheets, cells, shapes, or whatever else you have inside your workbook. This makes macros very easy to distribute. You simply send someone the Excel file and the macros are included.

All macros need an event to start them running. Earlier in this chapter, I discussed assigning a shortcut key to a macro you record. Pressing the shortcut key is one type of event that can trigger a macro. In this section, I discuss the Personal Macro Workbook, a special workbook for storing macros, and a few common ways to trigger macros.

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 This file, named personal.xlsb, doesn't exist until you record a macro using Personal Macro Workbook as the destination. The Personal Macro Workbook is loaded whenever you start Excel.

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 (refer to Figure 1-1).

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.



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

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 worksheets. 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 worksheet 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 on 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 worksheet, the Assign Macro dialog box, shown in Figure 1-8, opens 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.**

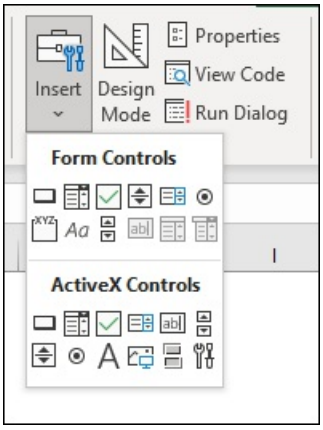


FIGURE 1-7: You can find the form controls on the Developer tab.

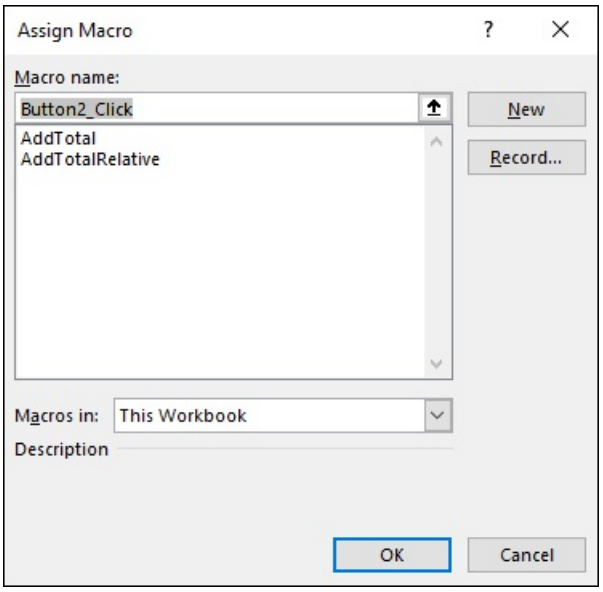


FIGURE 1-8: Assign a macro to the newly added button.

FORM CONTROLS VERSUS ACTIVEX CONTROLS

The Insert command on the Developer tab shows both form controls and ActiveX controls (refer to Figure 1-7). Although they look similar, they're quite different. Form controls are designed specifically for use on a worksheet, and ActiveX controls are typically used on Excel user forms (custom dialog boxes that are beyond the scope of this book). As a general rule, you should always use form controls when working on a worksheet unless there is a specific function that only ActiveX controls have. This is because 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) can 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 on the Quick Access Toolbar. The Quick Access Toolbar sits either above or below the Ribbon. You can add a custom button that runs your

macro by following these steps:

1. **Right-click your Quick Access Toolbar and select Customize Quick Access Toolbar.** This opens the dialog box shown in Figure 1-9.
2. **Select Macros from the Choose Commands From drop-down list on the left.**
3. **Select the macro you want to add and click the Add button.**
4. **(Optional) Click the Modify button to change the icon or display name, and then click OK to close the Modify Button dialog box.**
5. **Click OK to close the Excel Options dialog box.**

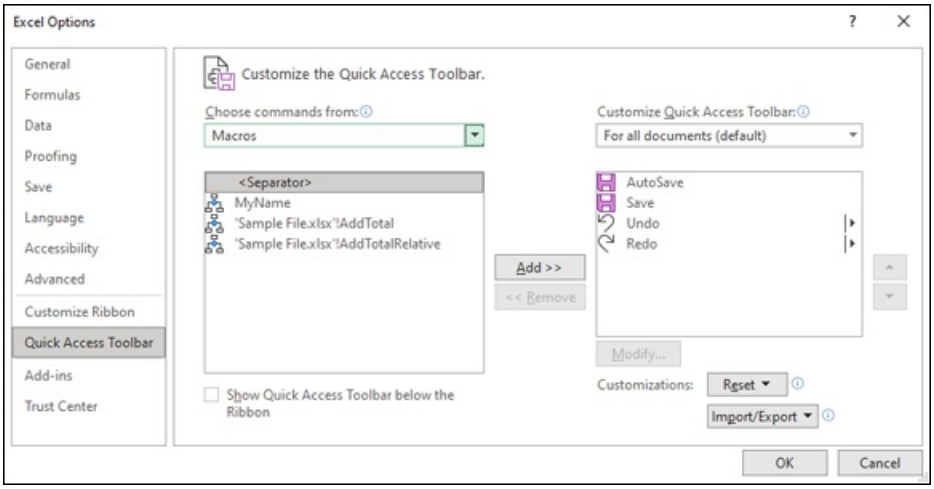


FIGURE 1-9: Adding a macro to the Quick Access Toolbar.

Exploring Macro Examples

Covering the fundamentals of building and using macros is one thing. Coming up with good ways to incorporate them into your reporting processes is another. Take a moment to review a few examples of how macros automate simple reporting tasks.



REMEMBER Open Chapter 1 Samples.xlsm to follow along in the next section.

Building navigation buttons

One common use of macros is navigation. Workbooks that have many worksheets or tabs can be frustrating to navigate. To help your users, you can create some sort of a switchboard, similar to the one shown in Figure 1-10. When a user clicks the Example 1 button, he's taken to the Example 1 sheet.

Creating a macro to navigate to a sheet is quite simple.

1. **Start at the sheet that will become your switchboard or starting point.**

2. **Start recording a macro.**
3. **While recording, click the destination sheet (the sheet this macro will navigate to).**
4. **After you click in the destination sheet, stop recording the macro.**
5. **Assign the macro to a button.**

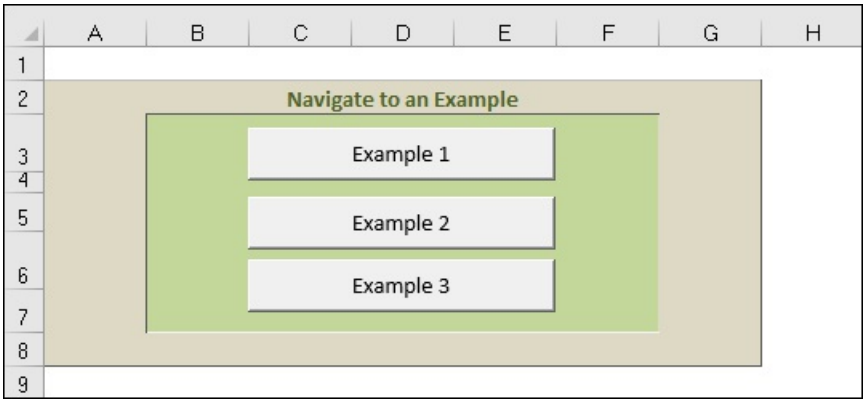


FIGURE 1-10: Use macros to build buttons that help users navigate your reports.



TIP It's useful to know that Excel has a built-in hyperlink feature, allowing you to convert the contents of a cell into a hyperlink that links to another location. That location can be a separate Excel workbook, a website, or even another tab in the current workbook. Although using a hyperlink may be easier than setting up a macro, you can't apply a hyperlink to form controls (such as buttons). Instead of a button, you'd use text to let users know where they'll go when they click the link.

Dynamically rearranging PivotTable data

Macros can be used with any Excel object normally used in reporting. For instance, you can use a macro to give your users a way to dynamically change aPivotTable. In the example shown in Figure 1-11, macros allow a user to change the perspective of the chart simply by clicking any one of the buttons.

Figure 1-12 reveals that the chart is actually a PivotChart tied to a PivotTable. The recorded macros assigned to each button are doing nothing more than rearranging the PivotTable to slice the data using various pivot fields.

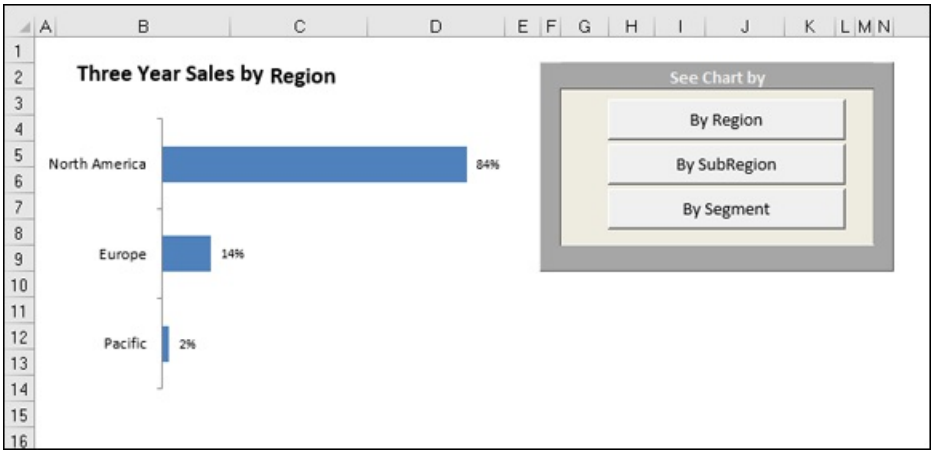


FIGURE 1-11: This report allows users to choose their perspective.

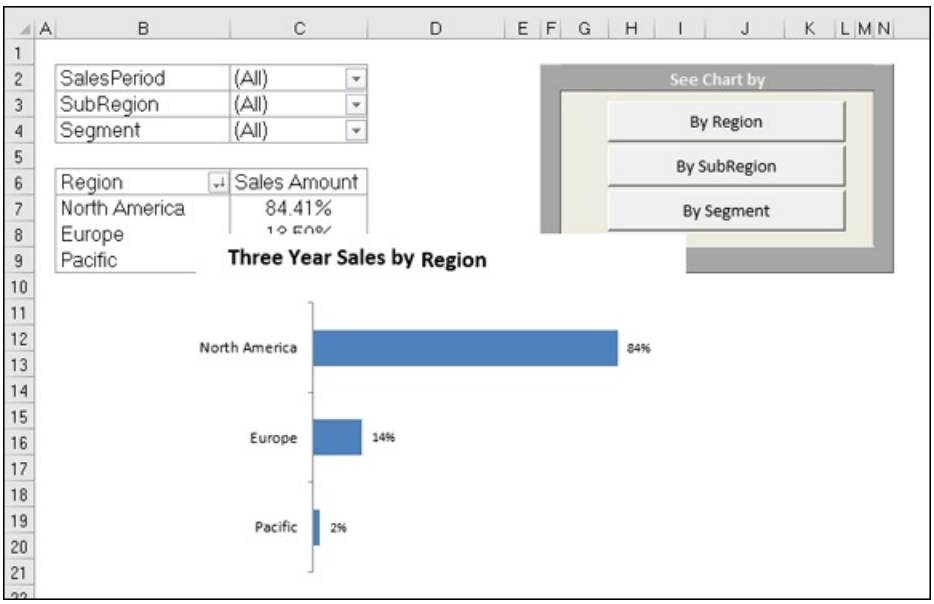


FIGURE 1-12: The macros behind these buttons rearrange the data fields in a PivotTable.

Here are the high-level steps needed to create this type of setup:

1. **Create your PivotTable and PivotChart.**
2. **Start recording a macro.**
3. **While recording, move a pivot field from one area of the PivotTable to the other. When you're done, stop recording the macro.**
4. **Record another macro to move the data field back to its original position.**
5. **After both macros are set up, assign each one to a separate button.**

You can run your new macros in turn to see your pivot field dynamically move back and forth.

Offering one-touch reporting options

The last two examples demonstrate that you can record any action that you find of value. That is, if you think users would appreciate a certain feature being automated for them, why not record a macro to do so?

In Figure 1-13, buttons are tied to macros that filter the PivotTable for the top or bottom 20 customers. Because the steps to filter a PivotTable for the top and bottom 20 have been recorded, users can get the benefit of this functionality without knowing how to do it themselves. Also, recording a specific action allows you to manage risk a bit. That is to say, your users can interact with your reports in a method that has been developed and tested by you.

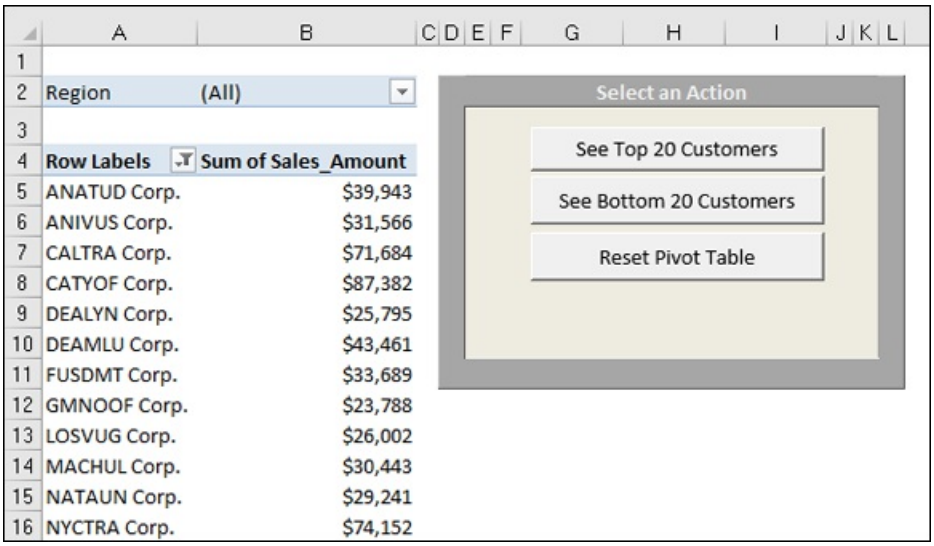


FIGURE 1-13: Offering prerecorded views not only saves time and effort, but it also allows users that don't know how to use advanced features to benefit from them.

This not only saves them time and effort, but it also allows users that don't know how to take these actions to benefit from them.

Figure 1-14 demonstrates how you can give your audience a quick and easy way to see the same data on different charts. It's not uncommon to be asked to see the same data different ways. Instead of taking up real estate by showing both charts, just record a macro that changes the Chart Type of the chart. Your clients can switch views to their heart's content.

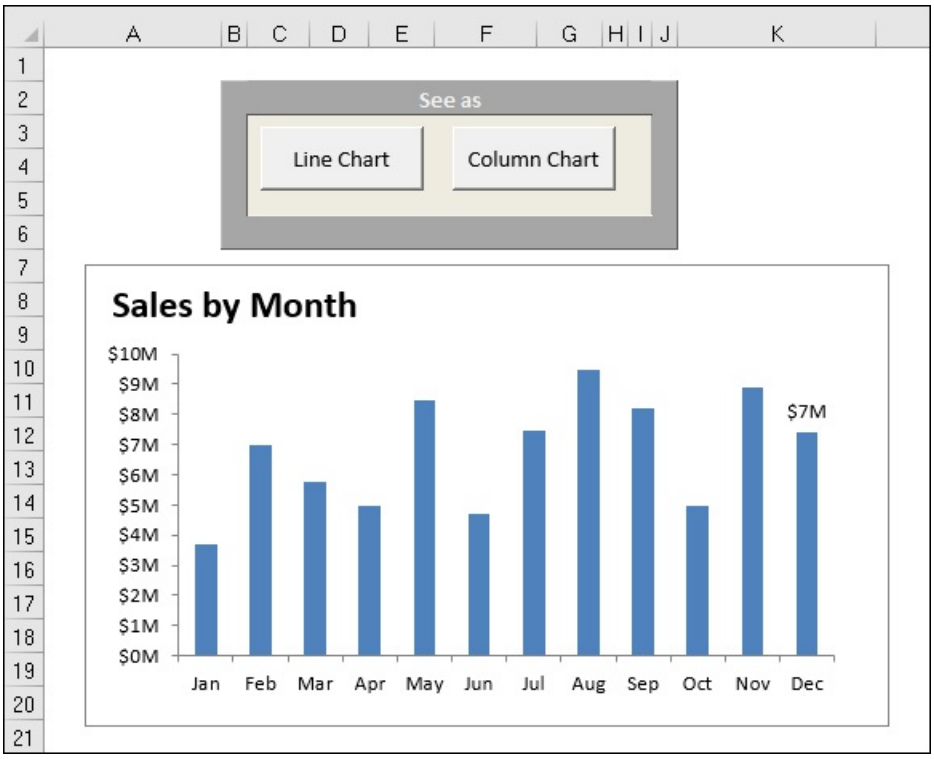


FIGURE 1-14: You can give your audience a choice in how they view data.

Chapter 2

Getting Cozy with the Visual Basic Editor

IN THIS CHAPTER

- **Understanding the Visual Basic Editor components**
- **Working with the Project Explorer**
- **Using a Code pane**
- **Customizing the Visual Basic Editor**

The Visual Basic Editor (VBE) is the environment where all Excel macros are written or recorded. The VBE is included with Excel free of charge. Even if you never record one macro, the VBE is in the background waiting to be used. When you create a macro, the VBE quietly comes to life ready to process the various procedures and routines you give it.

In this chapter, you take your first look behind the curtain to explore the VBE.

Working in the Visual Basic Editor

The VBE is actually a separate application that runs when you open Excel. To see this hidden VBE environment, you need to activate it. With Excel open, one of these two ways does the trick:

- Press Alt+F11.
- Click the Visual Basic button on the Ribbon's Developer tab.

To return to Excel, press Alt+F11 or click the Close button in the VBE.

Figure 2-1 shows the VBE with the key parts identified. Chances are your VBE program window won't look exactly like what's shown in Figure 2-1. The VBE contains several windows and is highly customizable. You can hide windows, rearrange windows, dock windows to the main window, and so on.

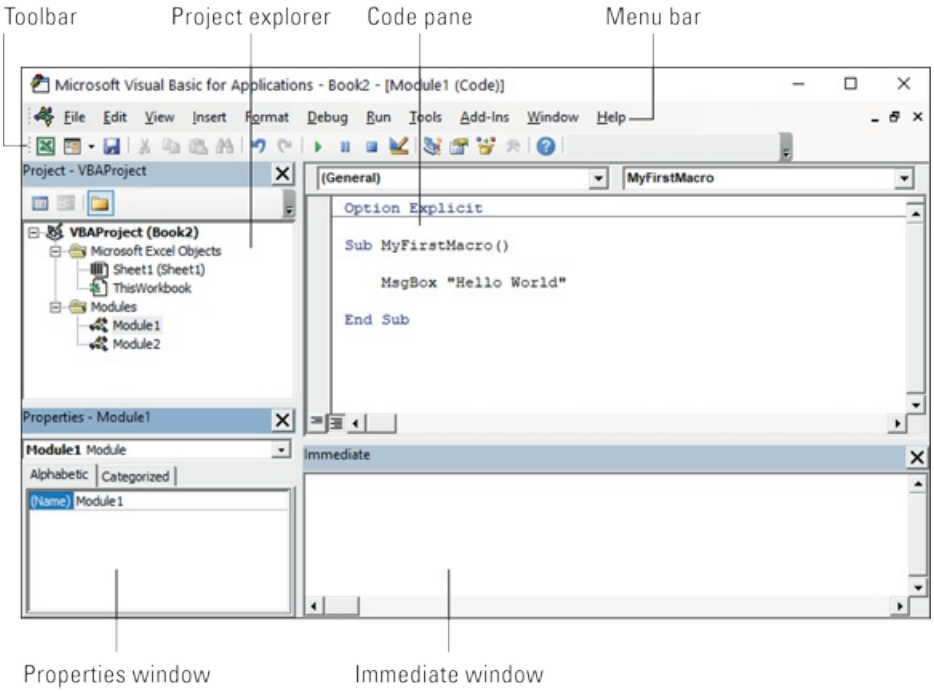


FIGURE 2-1: The VBE with significant elements identified.

The VBE 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. Many of the menu commands also have shortcut keys associated with them, such as Ctrl+G to show the Immediate window and Ctrl+R to show the Project Explorer.

The VBE also features shortcut menus. You can right-click virtually anything in the VBE and get a shortcut menu of common commands specific to the area you clicked. For example, clicking in the Project Explorer shows a different list of commands that clicking in Code pane.

The VBE toolbars

The VBE has four toolbars: Standard, Edit, Debug, and Userform. The Standard toolbar is the only one shown by default and is directly under the menu bar. It contains many of the most commonly used commands such as Return to Excel, Save, Cut, Copy, and Paste. You can customize the toolbars, move them around, display other toolbars, and so on. If you're so inclined, choose View ⇒ Toolbars to work with the VBE toolbars.

The Project Explorer

The Project Explorer 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. You explore this window in more detail in the "Working with the Project Explorer" section later in this chapter.

If the Project Explorer is not visible, press Ctrl+R or choose View ⇒ Project Explorer. To hide the Project Explorer, click the Close button in its title bar. Alternatively, right-click

anywhere in the Project Explorer and select Hide from the shortcut menu.

The Code pane

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

You find out more about Code panes later in this chapter's "Working with a Code Pane" section.

The Immediate window

The Immediate window may or may not be visible. If it isn't visible, press Ctrl+G or choose View ⇒ Immediate Window. 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 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. Collapse 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 collapse them.

Figure 2-2 shows the Project Explorer with two projects listed: a workbook named Book1 and a workbook named Book2.

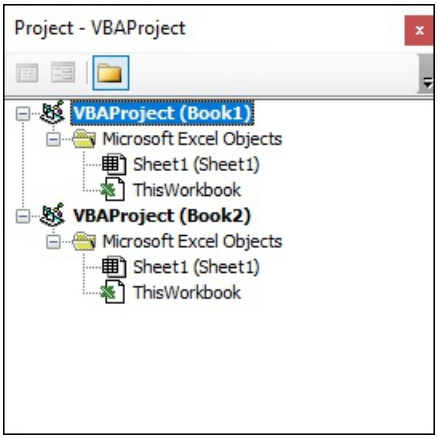


FIGURE 2-2: The Project Explorer with two projects open, expanded to show their objects.

Every project expands to show at least one folder called Microsoft Excel Objects. This folder expands to show an item for each sheet in the workbook (each sheet is considered an object),

and another object called ThisWorkbook (that represents the Workbook object). If the project has any VBA modules, the project listing also shows a Modules folder.

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 choose to store the recorded macro, just before you start 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 module-level variables (variables that apply to all procedures in the module instead of just one), or set some other module-wide options.
- **Sub procedures:** A set of programming instructions that performs some action. All recorded macros are 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 dependent on how may macros you have. If you have just a few, one module is probably all you need. If you start to get a lot of macros in a workbook, splitting them into well-named folders is best so you can easily find them later. Simply cut and paste any code from one module to another to move it.

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

1. **Select the project's name in the Project Explorer.**
2. **Choose Insert ⇒ Module.**

Or you can

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

The new module is added to a Modules folder in the Project Explorer (see Figure 2-3). Any modules you create in a given workbook are placed in this Modules folder.

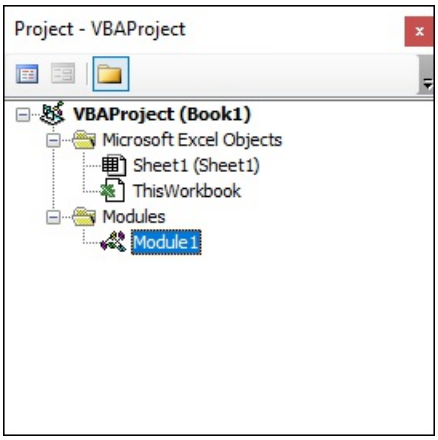


FIGURE 2-3: Code modules are visible in the Project Explorer in a folder called Modules.

If you want to change the name of your module, select the module in the Project Explorer and press F4 to show the Properties window. Modules only have one property, Name, so it's easy to find where to change it.

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 Explorer.**
2. **Choose File ⇒ Remove *xxx*, where *xxx* is the module name.**

Or

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

Whichever method you choose, Excel asks you if you want to export the module before removing it. Click Yes to create an export file that you can re-import into your project if you find you deleted it in error.



REMEMBER 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 Pane

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

Minimizing and maximizing windows

Code panes 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 pane that they're working on. Doing so lets you see more code and keeps you from getting distracted.

To maximize a Code pane, 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 pane to its original size, click the Restore button.

Sometimes, you may want to have two or more Code panes 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 panes manually, or choose Window ⇒ Tile Horizontally or Window ⇒ Tile Vertically to arrange them automatically.

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

Minimizing a Code pane gets it out of the way. You can also click the pane's Close button in the title bar to close it 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 Explorer. Working with these Code panes 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.

Chapter 1 shows you how to create 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 line-continuation 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 delete a statement that you shouldn't have, click the Undo button on the Standard toolbar (or press Ctrl+Z) until the statement appears again. After undoing, you can click 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 open the VBE.**
3. **Click the new workbook's name in the Project Explorer.**
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.**



TIP The VBE has its own set of shortcut keys you can use to quickly run a command using your keyboard. F5 is a shortcut for Run ⇒ Run Sub/UserForm.

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)



TIP As 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 reentering the code, you can open 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 VBE

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. The Options dialog box has 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 2-4 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.

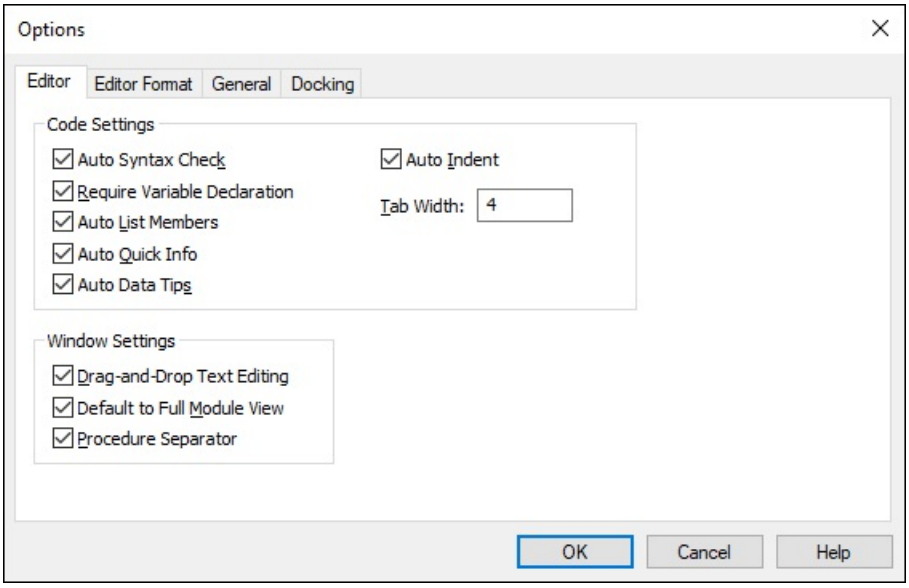


FIGURE 2-4: The Editor tab in the Options dialog box.

The Auto Syntax Check option

The Auto Syntax Check setting determines whether the VBE opens 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 (red by default) from the rest of the code, and you don't have to deal with any onscreen dialog boxes.

The Require Variable Declaration option

If the Require Variable Declaration option is set, VBE inserts the following statements 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.

If you don't set this option, VBA won't require you to explicitly declare your variables. If you don't declare a variable, VBA will declare it for you the first you use it. That is, VBA will create a spot in memory with that variable's name. That may seem like a time saver, but the first time you mistype a variable, you'll be glad you required variable declaration. Bugs from misspelled variables can be hard to find.

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 feature is very useful for saving time while coding.

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. There is 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. Most Excel developers are keen on using indentations in their code, so this option is typically kept on.



TIP 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.

To show the VBE's Edit toolbar, right-click any toolbar and select Edit from the context menu. It 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.

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 pane appear as a single scrollable list. If this option is turned off, you can see only one procedure at a time.

The Procedure Separator option

When the Procedure Separator option is turned on, separator bars appear between each procedure in a Code pane. Separator bars provide a nice visual line between procedures, making it easy to see where one piece of code ends and where another starts.

The Editor Format tab

Figure 2-5 shows the Editor Format tab of the Options dialog box. With this tab, you can customize the way the VBE looks.

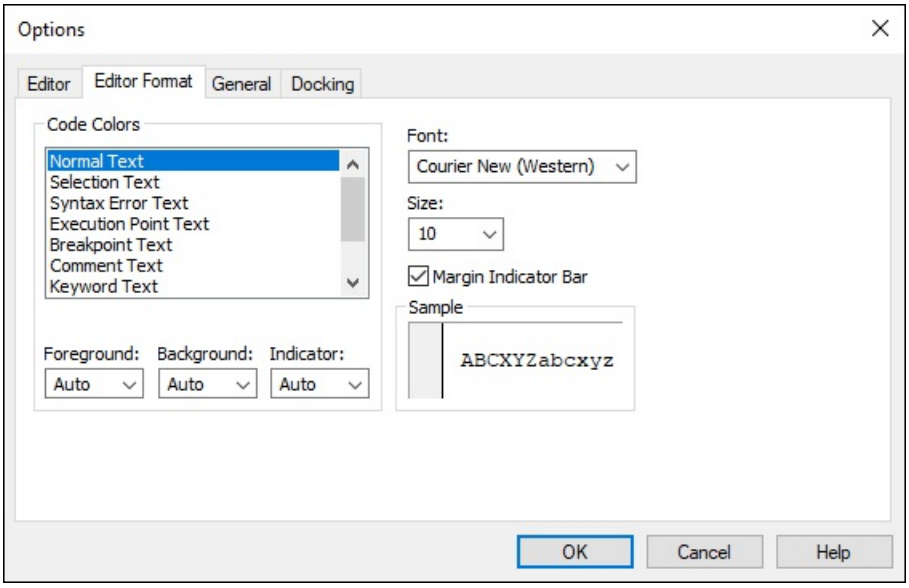


FIGURE 2-5: Change the way VBE's looks with the Editor Format tab.

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.Some Excel developers stick with the default colors. But if you like to change things up, you can 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 2-6 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. If you are just starting your Excel macro writing career, it's best to leave the Error Trapping set to Break on Unhandled

Errors. If Excel encounters an error, it stops executing, highlights the line that caused the error, and gives you a chance to figure out what went wrong.

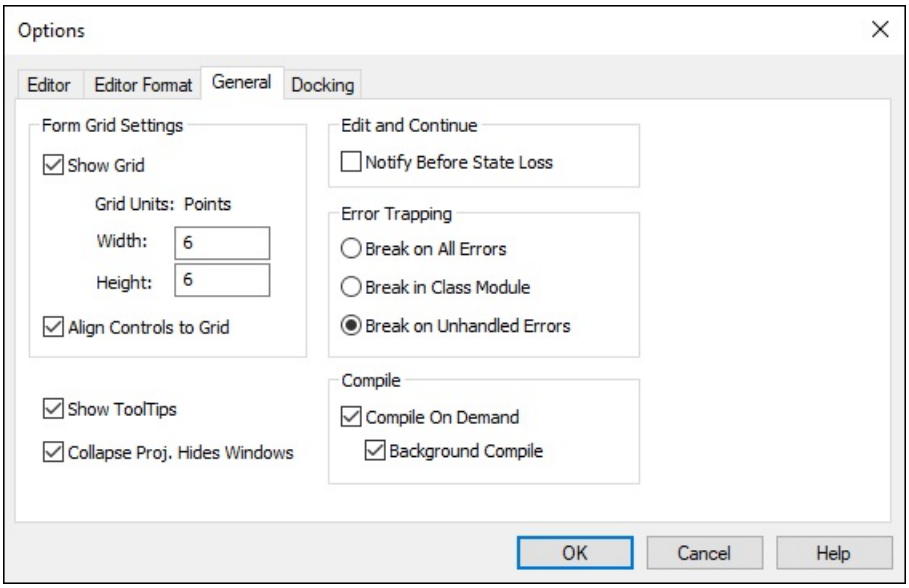


FIGURE 2-6: The General tab of the Options dialog box.

The Docking tab

Figure 2-7 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.

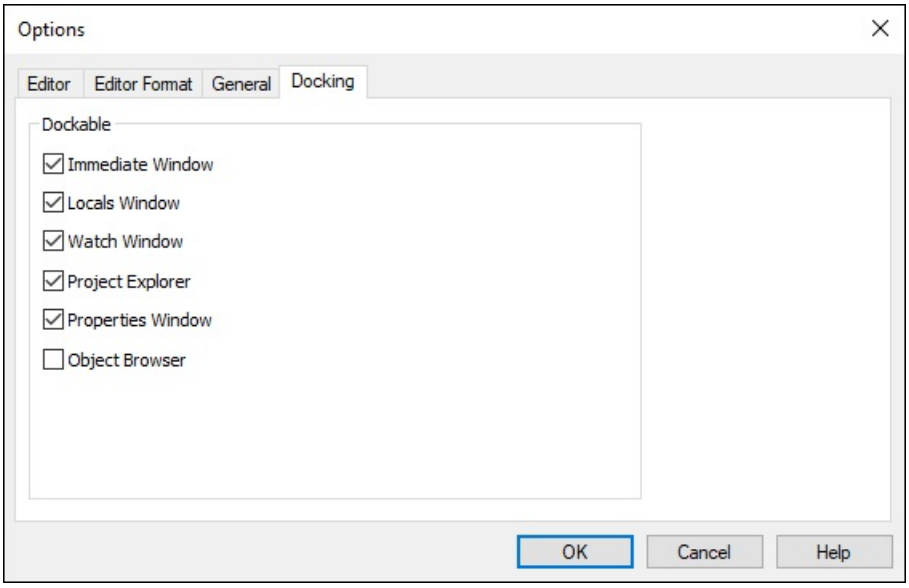


FIGURE 2-7: The Docking tab of the Options dialog box.

Chapter 3

The Anatomy of Macros

IN THIS CHAPTER

- **Understanding the Excel object model**
- **Understanding variables**
- **Understanding event procedures**
- **Handling errors in a nutshell**

The engine behind macros is Visual Basic for Applications (VBA). When you record a macro, Excel is busy writing the associated VBA behind the scenes. To fully understand macros, it's important to understand the underlying VBA typically used in Excel macros.

This chapter starts you on that journey by giving you a primer on some of the objects, variables, events, and error handlers you encounter in macros.

A Brief Overview of the Excel Object Model

VBA is an object-oriented programming language. The basic concept of object-oriented programming is that a software application (Excel in this case) consists of various individual objects, each of which has its own set of characteristics and uses. An Excel application contains workbooks, worksheets, cells, charts, PivotTables, drawing shapes — the list of Excel's objects is seemingly endless. Each object has its own set of characteristics, which are called *properties*, and its own set of uses, called *methods*.

You can think of this concept just as you would the objects you encounter every day, such as your computer in your office, your car in your garage, or the refrigerator in your kitchen. Each of those objects has identifying qualities, such as height, weight, and color. They each have their own distinct uses, such as your computer for working with Excel, your car to transport you over long distances, and your refrigerator to keep your perishable foods cold.

VBA objects also have their identifiable properties and methods of use. A worksheet cell is an object, and among its describable features (its properties) are its address, its height, its formatted fill color, and so on. A workbook is also a VBA object, and among its usable features (its methods) are its abilities to be opened, closed, and have a chart or PivotTable added to it.

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 structured according to the relationships between them.

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 using the dot operator. For example, to get to cell A1 on Sheet 1, you can enter this code:

```
Activeworkbook.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 already selected 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 a 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 called MySheet in a particular workbook that is open but not active, you must qualify the worksheet reference and the workbook reference, as follows:

```
Workbooks("MyData.xlsx").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 — such as the color of your house. Other properties cannot be changed — such as the age of your house. (The age of your house does change, but you physically can't change it. Only time can do that.)

Likewise, an object in Excel such as 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 example, 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 example, the Text property of a cell. The Text property gives you the formatted appearance of value in a cell, but you cannot overwrite or change it. To change the Text property, you have to change the Value property, the NumberFormat property, or both.

Understanding methods

Methods are the actions that you can perform 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 dictate how the method is applied. For example, you can use the Paste method more effectively by explicitly defining the Destination parameter:

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

A Brief Look at Variables

Another concept you see throughout macros is the concept of *variables*. It’s important to dedicate a few words to this concept, as 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.

The common variable types

Some of the common types of variables you use in macros are

- **String:** Holds text
- **Long:** Holds numbers ranging from -2,147,483,648 to 2,147,483,647
- **Double:** Holds floating point numbers
- **Variant:** Holds any kind of data
- **Boolean:** Holds True or False
- **Object:** Holds an 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 (abbreviation for dimension), then the name of your variable, then the type. For example:

```
Dim MyText as String

Dim MyNumber as Long

Dim MyWorksheet as Worksheet
```

Once you create your variable, you can fill it with data. Here are a few simple examples of how you would create a variable, and then assign values to them.

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

Dim MyNumber as Long
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 create. It all depends on the task at hand. As you work more with macros, values will make a lot more sense.

While it is possible to create code that does not use variables, many examples of VBA code do use variables. There are two main reasons for this.

First, Excel doesn’t inherently know what your data is used for. It doesn’t see numbers, symbols, or letters. It just sees data. When you declare variables with specific data types, you help Excel know how it should handle certain pieces of data so that your macros produce the results you'd expect.

Secondly, variables help make your code more efficient and easier to understand. For example, suppose you have a number in cell A1 that you are repeatedly referring to in your macro. You could retrieve that number by pointing to cell A1 each time you need it.

```
Sub Macro1()

  Range("B1").Value = Range("A1").Value * 5

  Range("C1").Value = Range("A1").Value * 10

  Range("D1").Value = Range("A1").Value * 15

End Sub
```

However, this would force Excel to waste cycles storing the same number into memory every time you point to cell A1. Also, if you need to change your workbook so that the target

number is not in cell A1, but in, say, cell A2, you would need to edit your code by changing all the references from A1 to A2.

A better way is to store the number in cell A1 just once. For example, you can store the value in cell A1 in an Long variable called myValue.

```
Sub WithVariable()

  Dim myValue As Long

  myValue = Range("A1").Value

  Range("C3").Value = myValue * 5

  Range("D5").Value = myValue * 10

  Range("E7").Value = myValue * 15

End Sub
```

This not only improves the efficiency of your code (ensuring Excel reads the number in cell A1 just once), it also ensures that you only have to edit one line should the design of your workbook change.

Understanding Event Procedures

Many macros implement code as *event procedures*. To fully understand event procedures, it's important to get acquainted with events.

An *event* is nothing more than an action that takes place during a session in Excel. Everything that happens in Excel happens to an object through an event. A few examples of events are opening a workbook, adding a worksheet, changing a value in a cell, saving a workbook, double-clicking a cell, and the list goes on.

The nifty thing is that you can tell Excel to run a certain macro or piece of code when a particular event occurs. For example, you may want to ensure that your workbook automatically saves each time it closes. You can add code to the BeforeClose workbook event that saves the workbook before it closes.



REMEMBER Chapter 2 discusses how to create a new VBA module to hold the code you write. However, event procedures are special in that they are not stored in the standard modules. Event procedures are stored directly within each object's built-in modules, which the next few sections explain.

Worksheet events

Worksheet events occur when something happens to a particular worksheet, such as when a worksheet is selected, when a cell on the worksheet is edited, or when a formula on a worksheet is calculated. Each worksheet has its own built-in module where you can place your own event procedures.

To get to this built-in module, you can right-click the worksheet's tab and select View Code

(see Figure 3-1).

The Visual Basic Editor automatically opens to the built-in module for the worksheet. This module's Code pane has two drop-down boxes at the top.

Select the Worksheet option from the drop-down list on the left. This action automatically selects the SelectionChange event from the drop-down list on the right. Some starter code is also added where you can enter or paste your code, as shown in Figure 3-2.

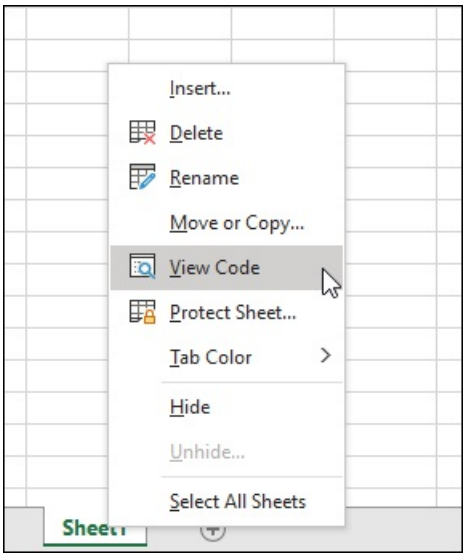


FIGURE 3-1: Getting to the built-in module for a worksheet.

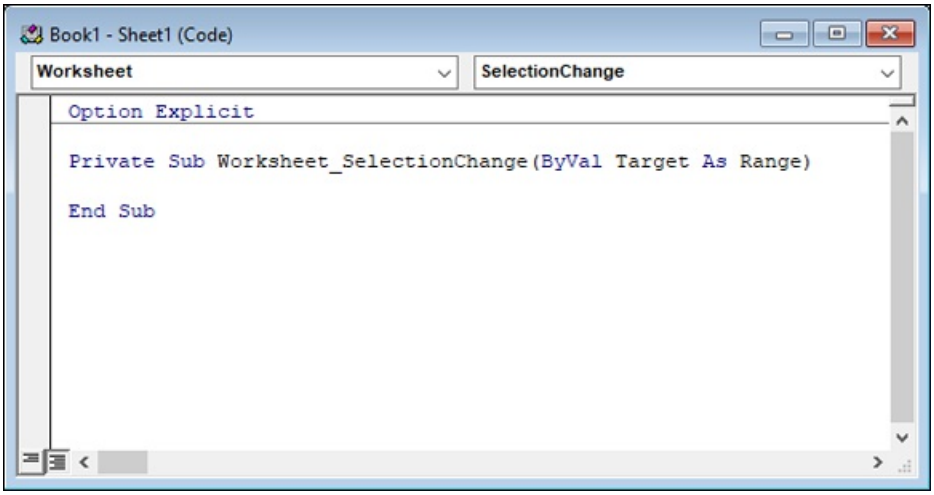


FIGURE 3-2: The default SelectionChange event for the Worksheet object.

The idea is to choose the most appropriate event from the Event drop-down menu for the task at hand. Figure 3-3 illustrates the different events you can choose.

The more commonly used worksheet events are

- **Worksheet_Change:** Triggers when any data on the worksheet is changed
- **Worksheet_SelectionChange:** Triggers each time a new cell or object on the worksheet

is selected

- **Worksheet_BeforeDoubleClick:** Triggers before Excel responds to a double-click on the worksheet
- **Worksheet_BeforeRightClick:** Triggers before Excel responds to a right-click on the worksheet
- **Worksheet_Activate:** Triggers when the user moves from another worksheet to this worksheet

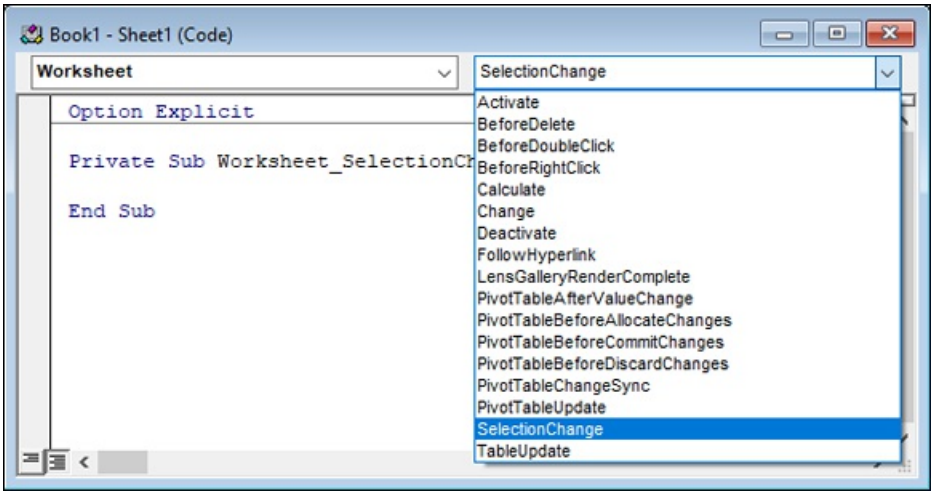


FIGURE 3-3: Click the Event drop-down list to choose the most appropriate event.

- **Worksheet_Deactivate:** Triggers when the user moves from this worksheet to another worksheet
- **Worksheet_Calculate:** Triggers each time a change on the worksheet causes Excel to recalculate formulas

Workbook events

Workbook events occur when something happens to a particular workbook, such as when a workbook is opened, when a workbook is closed, when a new worksheet is added, or when a workbook is saved. Each workbook has its own built-in module where you can place your own event procedures.

To get to this built-in module, you first need to activate the Visual Basic Editor (press Alt+F11). Then in the Project Explorer, double-click ThisWorkbook to open its Code pane. This module's Code pane has two drop-down boxes at the top.

Select the Workbook option from the drop-down list on the left. This action automatically selects the Open event from the drop-down list on the right. Some starter code is also added where you can enter or paste your code, as shown in Figure 3-4.

Choose the most appropriate event from the Event drop-down list for the task at hand. Figure 3-5 illustrates some of the events you can choose.

The more commonly used Workbook events are

- **Worksheet_Open:** Triggers when the workbook is opened
- **Worksheet_BeforeSave:** Triggers before the workbook is saved
- **Worksheet_BeforeClose:** Triggers before Excel closes the workbook
- **Worksheet_SheetChange:** Triggers when a user switches between sheets

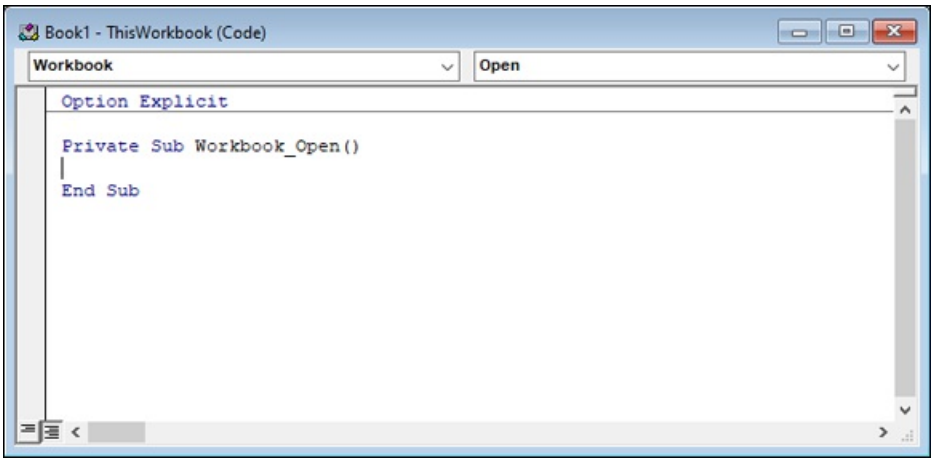


FIGURE 3-4: The default Open event for the Workbook object.

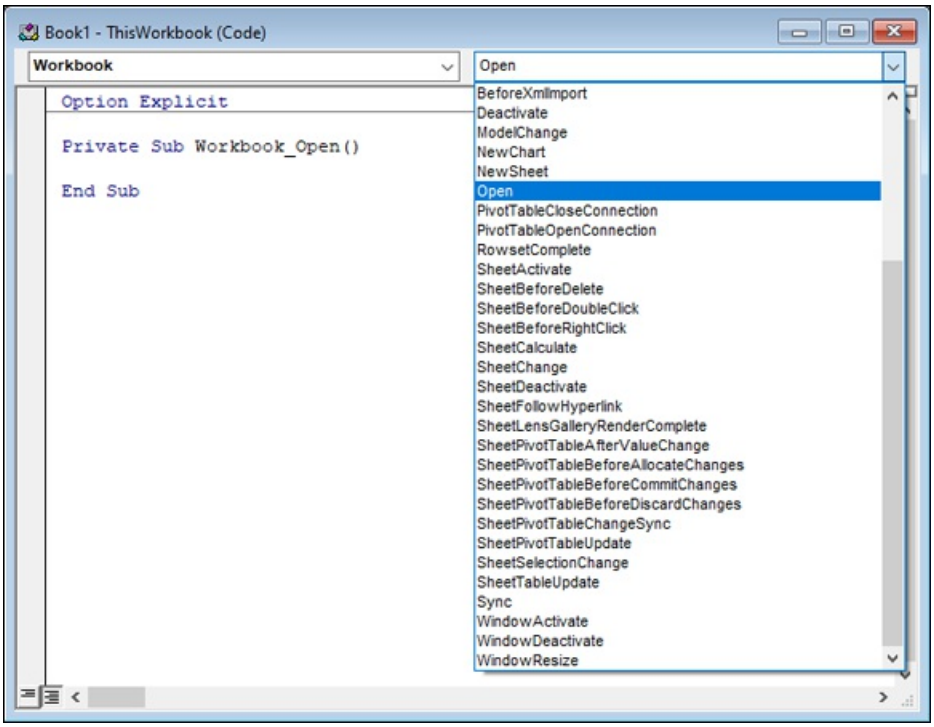


FIGURE 3-5: Click the Event drop-down list to choose the most appropriate event.

Error Handling in a Nutshell

In some macros, you see a line similar to this:

```
On Error GoTo MyError
```

This is called an *error handler*. Error handlers allow you to specify what happens when an error is encountered while your code runs.

Without error handlers, any error that occurs in your code prompts Excel to display a less-than-helpful error message and throws your users into the VBE to debug it. However, with the aid of error handlers, you can choose to ignore the error or exit the code gracefully with your own message to the user.

There are three types of On Error statements:

- **On Error GoTo *SomeLabel*:** The code jumps to the specified label.
- **On Error Resume Next:** The error is ignored and the code resumes.
- **On Error GoTo 0:** VBA resets to normal error-checking behavior.

On Error GoTo SomeLabel

There are times when an error in your code means you need to gracefully exit the procedure and give your users a clear message. In these situations, you can use the On Error GoTo statement to tell Excel to jump to a certain line of code.

Take this small piece of code for example. Here,it's telling Excel to divide the value in cell A1 by the value in cell A2, and then place the answer in cell A3. Easy. What could go wrong?

```
Sub CalcDivision()

  Range("A3").Value = Range("A1").Value / Range("A2").Value

End Sub
```

As it turns out, two major things can go wrong. If cell A2 contains a 0, you get a divide by 0 error. If cell A2 contains a non-numeric value, you get a type mismatch error.

To avoid a nasty error message, you can tell Excel that On Error, you want the code execution to jump to the label called ErrExit.

The following block of code includes the ErrExit label followed by a message to users that gives them friendly advice rather than a nasty error message. Also note the Exit Sub line before the ErrExit label. This ensures that the code simply exits if no error is encountered.

```
Sub CalcDivision()

  On Error GoTo ErrExit

  Range("A3").Value = Range("A1").Value / Range("A2").Value
  Exit Sub

ErrExit:
  MsgBox "Please Use Valid Non-Zero Numbers"

End Sub
```

On Error Resume Next

Sometimes, you want Excel to ignore an error and simply resume running the code. In these situations, you can use the On Error Resume Next statement.

For example, this piece of code is meant to delete a file called GhostFile.exe from the

C:\Temp directory. After the file is deleted, a nice message box tells the user the file is gone.

```
Sub DeleteGhost()

  Kill "C:\Temp\GhostFile.exe"

  MsgBox "File has been deleted."

End Sub
```

It works great if there is indeed a file to delete. But if for some reason the file called GhostFile.exe does not exist in the C:\Temp directory, an error is thrown.

In this case, you don't care if the file is not there. You were going to delete it anyway. So you can simply ignore the error and move on with the code.

By using the On Error Resume Next statement, the code runs its course even if the targeted file doesn't exist.

```
Sub DeleteGhost()

  On Error Resume Next

  Kill "C:\Temp\GhostFile.exe"

  MsgBox "File has been deleted."

End Sub
```

On Error GoTo 0

When using certain error statements, it may be necessary to reset the error-checking behavior of VBA. To understand what this means, take a look at this example.

Here, you first want to delete a file called GhostFile.exe from the C:\Temp directory. To avoid errors that may stem from the fact that the targeted file does not exist, you use the On Error Resume Next statement. After that, you are trying to do some suspect math by dividing 100/Mike.

```
Sub DeleteGhost()

On Error Resume Next

Kill "C:\Temp\GhostFile.exe"

Range("A3").Value = 100 / "Mike"

End Sub
```

Running this piece of code should generate an error due to the fuzzy math, but it doesn't. With On Error Resume Next you told Excel to ignore all errors, and that's what it does.

To remedy this problem, you can use the On Error GoTo 0 statement to resume the normal error-checking behavior.

```
Sub DeleteGhost()

  On Error Resume Next

    Kill "C:\Temp\GhostFile.exe"

  On Error GoTo 0

  Range("A3").Value = 100 / "Mike"

End Sub
```

This code ignores errors until the On Error GoTo 0 statement. After that statement, the code goes back to normal error checking where it triggers the expected error stemming from the fuzzy math.

Part 2

Making Short Work of Workbook Tasks

IN THIS PART …

Look at various techniques you can use to manipulate your workbooks.

See how you can leverage macros to automate the creation and duplication of Excel files.

Uncover macros that automate common worksheet tasks.

Explore how you can use macros to protect and back up your Excel workbooks.

Chapter 4
Working with Workbooks

IN THIS CHAPTER

- **Creating a new workbook from scratch**
- **Saving a workbook when a particular cell is changed**
- **Saving a workbook before closing**
- **Protecting an existing workbook on close**
- **Unprotecting an existing workbook on open**
- **Opening a workbook to a specific tab**
- **Opening a specific workbook defined by the user**
- **Determining whether a workbook is already open**
- **Determining whether a workbook exists in a directory**

A workbook is not just an Excel file; it's also an object in Excel's object model (a programming hierarchy that exposes parts of Excel to VBA).

This means that you can reference workbooks through VBA to do cool things such as automatically create new workbooks, prevent users from closing workbooks, and automatically back up workbooks.

In this chapter, you explore a few of the more useful workbook-related macros.

Installing Macros

You can install the macros in this chapter in three places depending on what kind of macro it is and how available you want the macro to be. Event macros are put into the module of the object the event is tied to. Non-event macros go into a standard module and can be in the workbook in which they are meant to run or in a workbook called the Personal Macro Workbook if you want to be able to run them on any workbook.

Event macros

In VBA, objects have events. VBA “listens” for these events to happen and runs special event macros when it detects and event. For example, the workbook object has an Open event. When VBA detects that the workbook has been opened, it looks for a procedure called Workbook_Open in the ThisWorkbook module. If it's there, it runs that macro.

Most of the event macros that you'll use, and most of the examples in the book, are workbook or worksheet events. Workbook event macros go in the ThisWorkbook module and worksheet event macros go in the module for the sheet you want the event code to run on.

To install an event macro, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **In the Project Explorer, find your project/workbook name and click the plus sign next to it to see all the objects.**
3. **Double-click ThisWorkbook.**
4. **Select the event in the right drop-down list.**
5. **Type or paste the code in the module.**

Personal Macro Workbook

There is a special workbook called the Personal Macro Workbook named personal.xlsb. If you record a macro, you can choose this workbook as the destination of the recorded macro. The workbook doesn't exist until you first record a macro into it. After that, it lives in a folder and Excel loads that workbook every time it starts.

The benefit of a workbook that opens every time Excel starts is that all the macros in that workbook are always available to you. If you have a macro that works on the active sheet or whatever range is currently selected, that's a good candidate for the Personal Macro Workbook.

To install a macro in the Personal Macro Workbook, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click personal.xlb in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**



REMEMBER If you don't see personal.xlsb in the Project Explorer, record a macro to the Personal Macro Workbook and Excel will create it for you.

Standard macros

All the other macro examples in this book are installed in a specific workbook in a standard module. If it's not an event macro and it's not a macro you want in the Personal Macro Workbook, then install it using the following steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click the project/workbook name in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**

Creating a New Workbook from Scratch

You may sometimes want or need to create a new workbook in an automated way. For example, you may need to copy data from a table and paste it into a newly created workbook. The following macro copies a range of cells from the active sheet and pastes the data into a new workbook.

This macro is relatively intuitive as you read through the lines of the code.

```
Sub CopyToNewWorkbook()

'Step 1: Copy the data.
  Sheets("Example 1").Range("B4:C15").Copy

'Step 2: Create a new workbook.
  Workbooks.Add

'Step 3: Paste the data.
  ActiveSheet.Paste Destination:=Range("A1")

'Step 4: Turn off application alerts.
  Application.DisplayAlerts = False

'Step 5: Save the newly created workbook.
  ActiveWorkbook.SaveAs _
    Filename:="C:\MyNewBook.xlsx"

'Step 6: Turn application alerts back on.
  Application.DisplayAlerts = True

End Sub
```

1. Copies the data on the “Example 1” sheet ranging from cells B4 to C15.

   The thing to note here is that you are specifying both the sheet and the range by name. This practice is best when you are working with multiple workbooks that have multiple worksheets. Because the workbook isn’t explicity named, the macro uses the active workbook. You may wish to name the workbook so the code always uses the same one.

2. The Add method of the Workbook object creates a new workbook.

   When you add a workbook, the new workbook immediately gains focus, becoming the active workbook. This is the same behavior you would see if you were to add a workbook manually (File ⇒ New ⇒ Blank Document).

3. The Paste method sends the data you copied to cell A1 of the new workbook. Pay attention to the fact that the code refers to the ActiveSheet object.

4. The DisplayAlerts method is set to False, effectively turning off Excel’s warnings. You do this because in the next step of the code, you save the newly created workbook. You may run this macro multiple times, in which case Excel attempts to save the file multiple times.

   What happens when you try save a workbook multiple times? That’s right — Excel warns you that there is already a file out there with that name and then asks if you want to overwrite the previously existing file. Because your goal is to automate the creation of the new workbook, you want to suppress that warning.

5. Saves the file by using the SaveAs method. Note that you are entering the full path of the save location, including the final filename.

6. Turns the applications alerts on again. If you don’t, Excel continues to suppress all warnings for the life of the current session.

Saving a Workbook when a Particular Cell Is Changed

Sometimes, you may be working on data that is so sensitive that you need to save your workbook every time a particular cell or range of cells is changed. This macro allows you to define the range of cells that, when changed, forces the workbook to save.

In the example shown in Figure 4-1, you want the workbook to save when an edit is made to any of the cells in the range C5:C16.

	A	B	C	D
4				
5		January	26,263	
6		February	25,343	
7		March	52,149	
8		dsff	72,579	
9		May	38,635	
10		June	60,175	
11		July	32,305	
12		August	14,288	
13		September	71,787	
14		October	48,402	
15		November	71,850	
16		December	77,798	
17				

FIGURE 4-1: Changing any cell in range C5:C16 forces the workbook to save.

The secret to this code is the Intersect method. Because you don't want to save the worksheet when any old cell changes, you use the Intersect method to determine whether the target cell (the cell that changed) intersects with the range you have specified to be the trigger range (C5:C16 in this case).

The Intersect method returns one of two things: either a Range object that defines the intersection between the two given ranges, or nothing. So in essence, you need to throw the target cell against the Intersect method to check for a value of Nothing. At that point, you can make the decision whether to save the workbook.

```
Private Sub Worksheet_Change(ByVal Target As Range)

'Step 1: Does the changed range intersect?
  If Not Intersect(Target, Range("C5:C16")) Is Nothing Then

'Step 2: If there is an intersection, save the workbook.
    ActiveWorkbook.Save

'Step 3: Close out the If statement.
  End If

End Sub
```

1. Checks to see whether the target cell (the cell that has changed) is in the range specified

by the Intersect method. If it's not in the range, Intersect returns Nothing (a special keyword in VBA meaning there is no object). If it’s in the range, the Is Nothing comparison returns False and the Not operator changes False to True.

2. If there is an intersection, employs the Save method of the active workbook, overwriting the previous version.
3. Closes the If statement. Every time you use an If-Then block you must close it with End If.

This is an event macro, so you must install it in the worksheet's module, as shown in Figure 4-2. (See the “Installing Macros” section at the beginning of this chapter for how.)

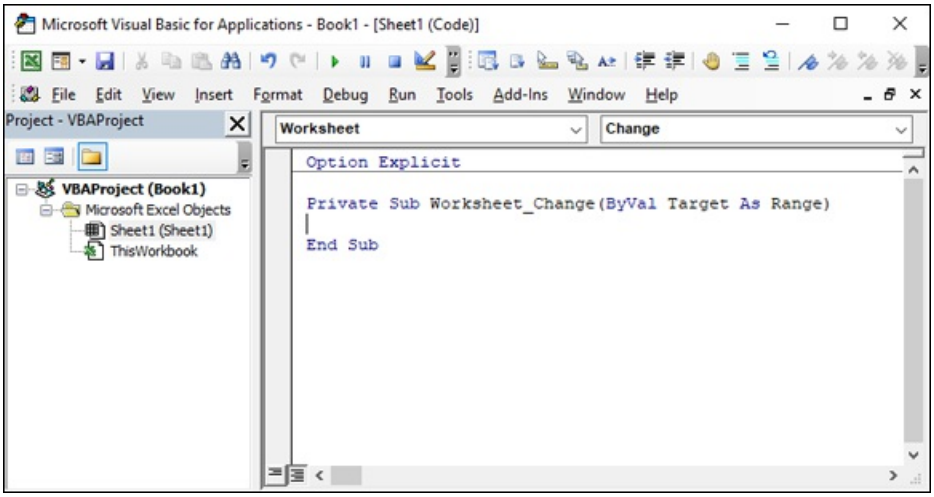


FIGURE 4-2: Enter or paste your code in the Worksheet_Change event Code pane.

Saving a Workbook before Closing

This macro is an excellent way to protect users from inadvertently closing their file before saving. When implemented, this macro ensures that Excel automatically saves before closing the workbook.



REMEMBER Excel normally warns users who are attempting to close an unsaved workbook, giving them an option to save before closing. However, many users may blow past the warning and inadvertently click No, telling Excel to close without saving. With this macro, you are protecting against this by automatically saving before close.

This code is triggered by the workbook's BeforeClose event. When you try to close the workbook, this event fires, running the code within. The crux of the code is simple — it asks the user whether they really wants to close the workbook (see Figure 4-3). The macro then evaluates whether the user clicked OK or Cancel.

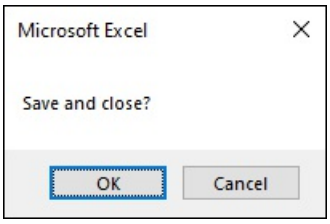


FIGURE 4-3: A message box opens when you attempt to close the workbook.

The evaluation is done with a Select Case statement. The Select Case statement is an alternative to the If…Then…Else statement, allowing you to perform condition checks in your macros. The basic construct of a Select Case statement is simple:

```
Select Case <some expression to check>
Case Is = <some value>
       <do something>
Case Is = <some other value>
       <do something else>
Case Is = <some 3rd value>
       <do some 3rd thing>
End Select
```

With a Select Case statement, you can perform many conditional checks. In this case, you are simply checking for OK or Cancel. Take a look at the code:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)

'Step 1: Activate the message box and start the check.
  Select Case MsgBox("Save and close?", vbOKCancel)

'Step 2: Cancel button pressed, cancel the close.
    Case Is = vbCancel
      Cancel = True

'Step 3: OK button pressed, save the workbook and close.
    Case Is = vbOK
      ActiveWorkbook.Save

'Step 4: Close your Select Case statement.
```

```
  End Select

End Sub
```

1. Shows the message box as the condition check for the Select Case statement. Here, the vbOKCancel argument displays an OK and Cancel buttons in the message.
2. If the user clicks Cancel in the message box, the Workbook_Close event is canceled. This is done by passing True to the Cancel Boolean. (This effectively cancels the event's action, preventing the workbook from closing.)
3. If the user clicks the OK button in the message box, workbook is saved. And because Cancel isn't set to True, Excel continues with the close.
4. Closes the Select Case statement. Every time you start a Select Case, you must close it with a corresponding End Select.

To install this macro, you need to copy and paste it into the Workbook_BeforeClose event Code pane, as shown in Figure 4-4. (See the "Installing Macros" section at the beginning of this chapter.) Placing the macro there allows it to run each time you try to close the workbook.

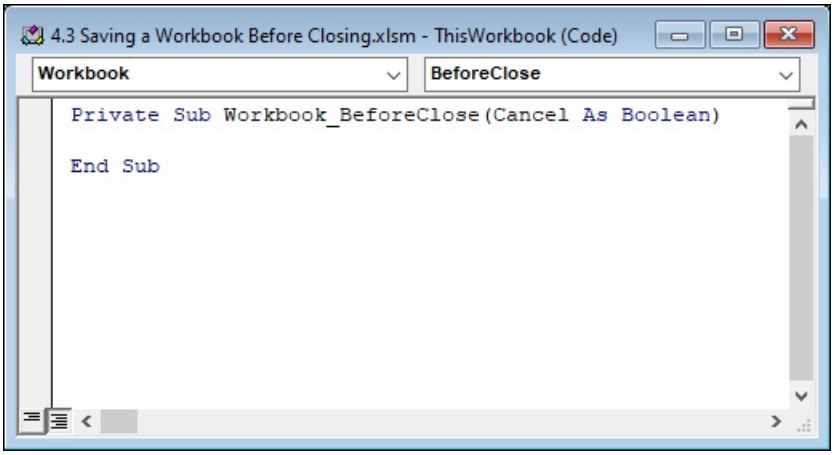


FIGURE 4-4: Type or paste your code in the Workbook_BeforeClose event Code pane.

Protecting a Worksheet on Workbook Close

Sometimes you need to send your workbook into the world with specific worksheets protected. If you find that you're constantly protecting and unprotecting sheets before distributing your workbooks, this macro can help you.

This code is triggered by the workbook's BeforeClose event. When you try to close the workbook, this event fires, running the code within. The macro automatically protects the specified sheet with the given password, and then saves the workbook.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)

'Step 1: Protect the sheet with a password.
  Sheets("Sheet1").Protect Password:="RED"

'Step 2: Save the workbook.
  ActiveWorkbook.Save

End Sub
```

1. Protects the sheet you want — Sheet1, in this case — and provides the password argument, Password:=RED. This defines the password needed to remove the protection.

   This password argument is completely optional. If you omit this altogether, the sheet is still protected, but you won't need a password to unprotect it.

   REMEMBER Also, Excel passwords are case-sensitive, so pay attention to the capitalization you are using.

2. Saves the workbook. If you don't save the workbook, the sheet protection you just applied won't be in effect the next time the workbook is opened.

To implement this macro, you need to copy and paste it into the Workbook_BeforeClose event Code window, as shown in Figure 4-5. (See the first section in this chapter, "Installing Macros.") Placing the macro here allows it to run each time you try to close the workbook.

Note that you can protect additional sheets by adding additional statements before the ActiveWorkbook.Save statement.

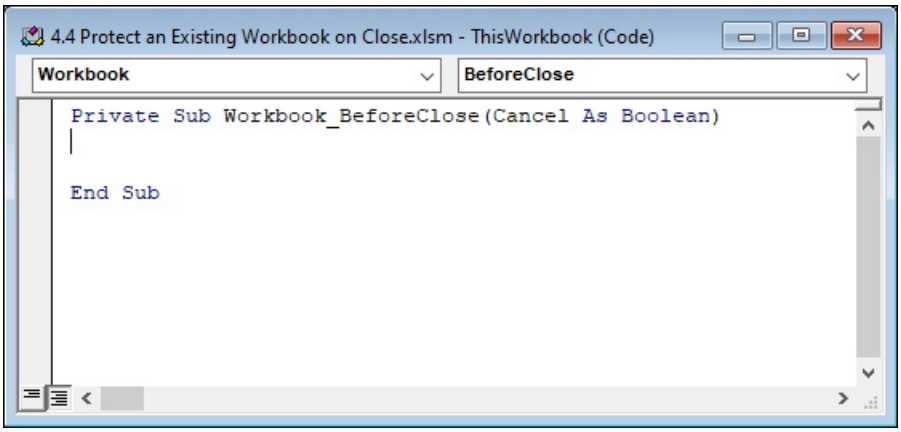


FIGURE 4-5: Type or paste your code in the Workbook BeforeClose event Code.

Unprotecting a Worksheet

If you've distributed workbooks with protected sheets, you likely get the workbooks back with the sheets still protected. While you may want those sheets protected for your users, you may need to unprotect the worksheets before continuing your portion of the work. If you find that you are continuously unprotecting worksheets, this macro may be just the ticket.

You can run this macro to automatically unprotect the specified sheet with the given password.

```
Sub UnprotectSheet()

'Step 1: Unprotect the sheet with a password.
  Sheets("Sheet1").Unprotect Password:="RED"

End Sub
```

The macro explicitly names the sheet you want to unprotect — Sheet1, in this case. Then it passes the password required to unprotect the sheet. Be aware that Excel passwords are case-sensitive, so pay attention to the exact password and capitalization that you are using.

To implement this macro, you can copy and paste it into a standard module, as shown in Figure 4-6. (See the "Installing Macros" section earlier in this chapter for details.)

You can unprotect additional sheets by adding additional statements.

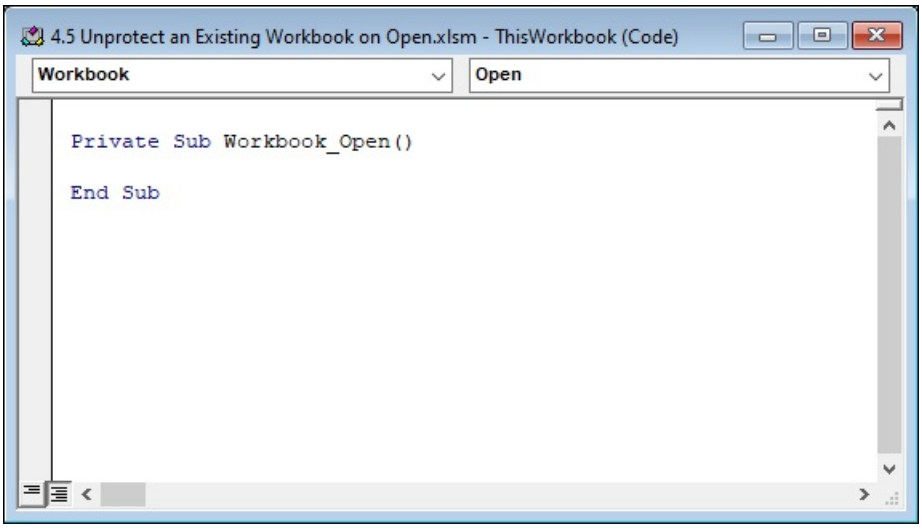


FIGURE 4-6: Type or paste your code in the Workbook Open event Code pane.

Opening a Workbook to a Specific Tab

In some situations, you may want a specific worksheet to be visible whenever your workbook is opened. With this macro, if a user is working with your workbook, they can't go astray because the workbook starts on the exact worksheet it needs to.

In the example shown in Figure 4-7, you want the workbook to go immediately to the sheet called Start Here.

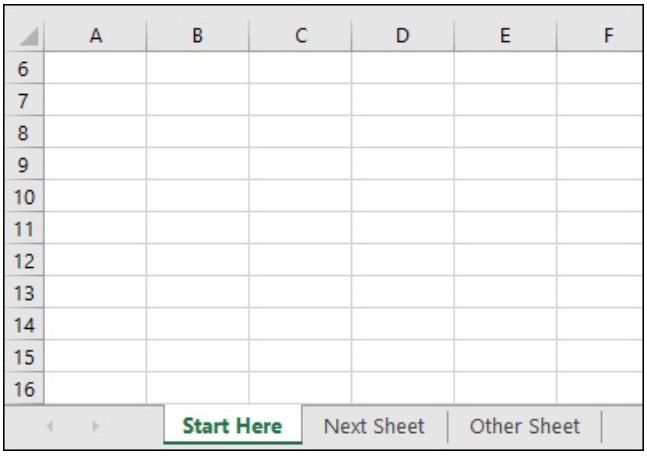


FIGURE 4-7: You want your workbook to automatically open to the sheet called Start Here.

This macro uses the workbook's Open event to start the workbook on the specified sheet when the workbook is opened.

```
Private Sub Workbook_Open()

'Step 1: Select the specified sheet.
  Sheets("Start Here").Select

End Sub
```

The macro explicitly names the sheet the workbook should jump to when it's opened.

To implement this macro, you need to copy and paste it into the Workbook_Open event code window, as shown in Figure 4-8. (See the earlier section, "Installing Macros" for more information.) Placing the macro here allows it to run each time the workbook opens.

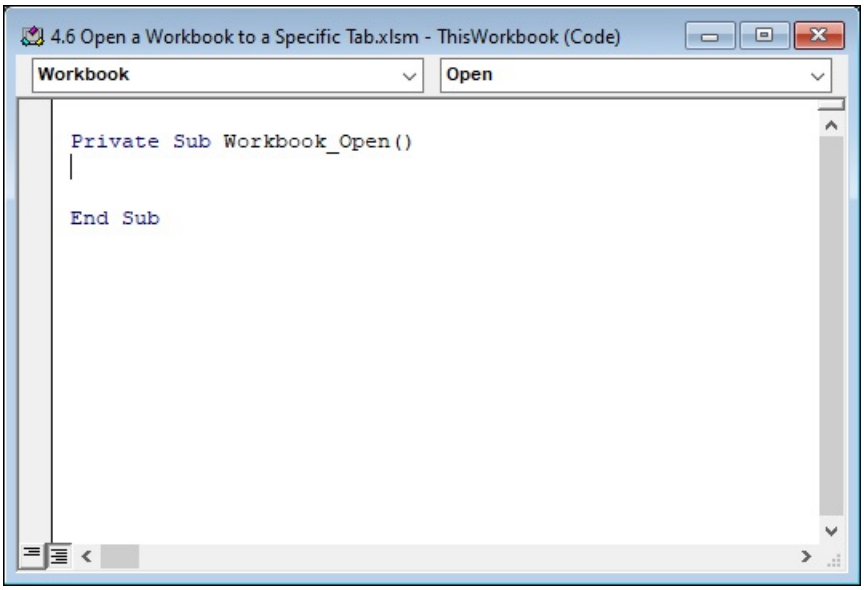


FIGURE 4-8: Type or paste your code in the Workbook Open event Code pane.

Opening a Specific Workbook Defined by the User

This macro uses the GetOpenFilename method to open a dialog box, allowing you to browse for and open the Excel file of your choosing.

This macro opens the dialog box shown in Figure 4-9, allowing the user to browse for and open an Excel file.

Here's how this macro works:

```
Sub OpenAWorkbook()

'Step 1: Define a variant variable.
  Dim FName As Variant

'Step 2: GetOpenFilename Method activates dialog box.
  FName = Application.GetOpenFilename( _
    FileFilter:="Excel Workbooks,*.xl*", _
    Title:="Choose a Workbook to Open", _
    MultiSelect:=False)

'Step 3: If a file was chosen, open it!
  If FName <> False Then
    Workbooks.Open Filename:=FName
  End If

End Sub
```

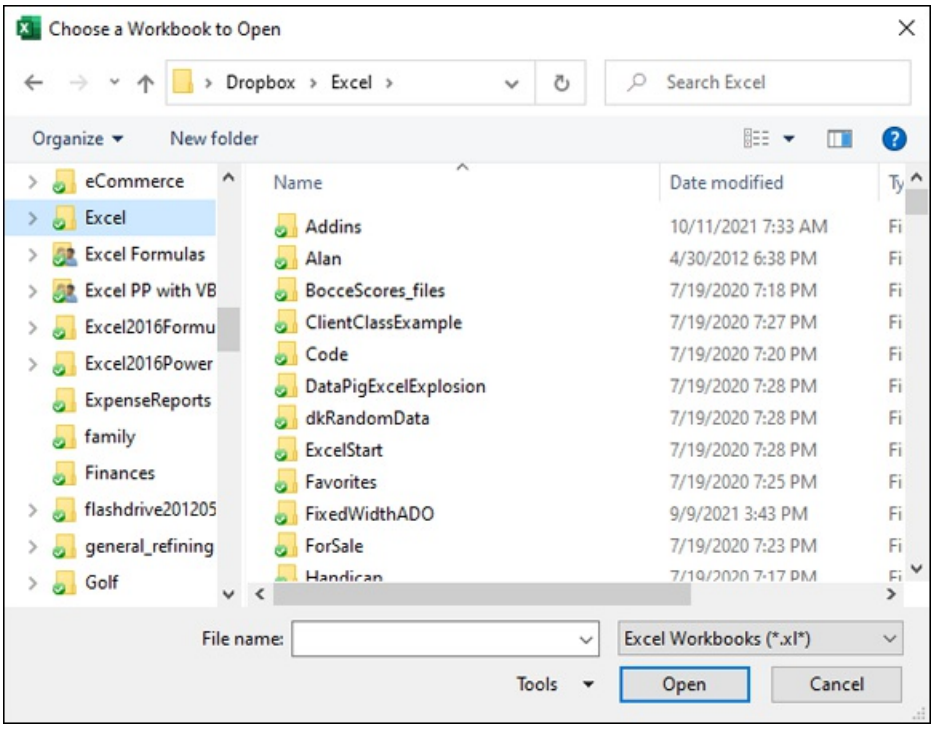


FIGURE 4-9: The macro opens the Open dialog box.

1. Declares a variant variable that holds the filename that the user chooses. FName is the name of the variable.

2. The GetOpenFilename method calls a dialog box that allows you to browse and select the necessary file.

   The GetOpenFilename method supports a few parameters. The FileFilter parameter specifies the type of file you are looking for. The Title parameter changes the title that appears at the top of the dialog box. The MultiSelect parameter limits the selection to one file when False or allows multiple files when True.

3. If the user selects a file from the dialog box, the FName variable is filled with the name of the file they chose. If the user clicks Cancel, the variable is set to False. If it's not False, the Open method of the Workbooks object opens the file.

To implement this macro, you can copy and paste it into a standard module. (See the earlier section, “Installing Macros,” for details.)

Determining Whether a Workbook Is Already Open

The previous macro automatically opened a workbook based on the user's selection. As you think about automatically opening workbooks, you must consider what may happen if you attempt to open a book that is already open. In the non-VBA world, Excel attempts to open the file again, with the message shown in Figure 4-10 warning that Excel can't have two workbooks open with the same name. You can protect against such an occurrence by checking whether a given file is already open before trying to open it again.

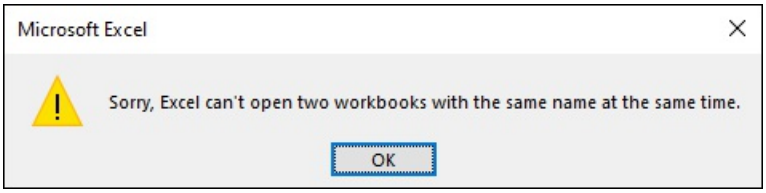


FIGURE 4-10: You can avoid this annoying message box when opening a workbook that is already open.

The first thing to notice about this macro is that it's a function, not a Sub procedure. With a function, you can pass a value to the procedure that calls the function. In this case, the function passes either True or False, depending on whether the workbook is open.

The gist of this code is simple: You are testing a given filename to see whether it can be assigned to an object variable. Only opened workbooks can be assigned to an object variable. When you try to assign a closed workbook to the variable, an error occurs.

So if the given workbook can be assigned, the workbook is open; if an error occurs, the workbook is closed.

```
Function FileIsOpen(TargetWorkbook As String) As Boolean

'Step 1: Declare variables.
  Dim TestBook As Workbook

'Step 2: Tell Excel to resume on error.
  On Error Resume Next

'Step 3: Try to assign the target workbook to TestBook.
  Set TestBook = Workbooks(TargetWorkbook)

'Step 4: If no error occurred then workbook is already open.
  If Err.Number = 0 Then
    FileIsOpen = True
  Else
    FileIsOpen = False
  End If

End Function
```

1. Declares a Workbook variable to hold the open workbook if it exists. TestBook is the name of the variable.
2. Tells Excel that an error may be running this code. In the event of an error, the error is ignored and the macro continues running the rest of the code. Without this line, the code

would simply stop when an error occurs.

3. Assigns the given workbook to the TestBook variable. The workbook you're trying to assign is itself a string variable called TargetWorkbook. TargetWorkbook is passed to the function in the function declarations (see the first line of the code). This structure eliminates the need to hard-code a workbook name, allowing you to pass it as a variable instead.
4. If no workbooks are open with that name, an error occurs. Excel ignores the error because of the statement in Step 2, but it does keep track that an error occurred, even if it was ignored. The Err object has a Number property that holds the error number. If it's zero, there was no error so a workbook with that name was open. If it's anything except zero, assigning the workbook to the variable failed. The function's name is set to True or False depending on the error number.

The following macro demonstrates how to implement this function. Here, you are using the same macro you saw in the previous section, "Opening a Specific Workbook Defined by the User," but this time, you are calling the new FileIsOpenTest function to make sure the user cannot open an already opened file.

```
Sub Macro1()

'Step 1: Define a string variable.
  Dim FName As Variant
  Dim FNFileOnly As String
'Step 2: GetOpenFilename method activates dialog box.
  FName = Application.GetOpenFilename( _
    FileFilter:="Excel Workbooks,*.xl*", _
    Title:="Choose a Workbook to Open", _
    MultiSelect:=False)

'Step 3: Open the chosen file if not already opened.
  If FName <> False Then
    FNFileOnly = Dir(FName)
    If FileIsOpen(FNFileOnly) = True Then
      Workbooks(FNFileOnly).Activate
    Else
      Workbooks.Open Filename:=FName
    End If
  End If

End Sub
```

This macro uses the GetOpenFilename method to allow the user to browse for a file. If the user clicks Cancel, the If statement that starts Step 3 skips over the rest of the statements. If the user selects a file, the Dir function is used to get only the filename and not the path. Then the function is called to determine whether the file is open. If it is, the workbook stays closed. If not, it's opened.

To implement this macro, you can copy and paste both pieces of code into a standard module. (See the earlier "Installing Macros" section at the beginning of this chapter.)

Determining Whether a Workbook Exists in a Directory

You may have a process that manipulates a file somewhere on your PC. For example, you may need to open an existing workbook to add new data to it on a daily basis. In these cases, you may need to test to see whether the file you need to manipulate actually exists. This macro allows you to pass a file path to evaluate whether the file is there.

The first thing to notice about this macro is that it is a function, not a sub procedure. Making this macro a function enables you to return True or False to the procedure that calls it.

The preceding example used the Dir function to strip away the path and leave only the filename. That was possible because you could be sure the file exists — the user just selected it. You can also use Dir to determine whether the file exists:

```
Function FileExists(FPath As String) As Boolean

'Step 1: Declare your variables.
  Dim FName As String

'Step 2: Use the Dir function to get the filename.
  FName = Dir(FPath)

'Step 3: If file exists, return True else False.
  If Len(FName) <> 0 Then
    FileExists = True
  Else
    FileExists = False
  End If

End Function
```

1. Declares a string variable to hold the filename that returned from the Dir function. FName is the name of the string variable.
2. Attempts to set the FName variable. Passes the FPath variable to the Dir function. This FPath variable is passed via the function declarations (see the first line of the code). This structure prevents you from having to hard-code a file path, passing it as a variable instead.
3. If the file exists, FName is the name of the file without the path. If it doesn't exist, FName is a zero length string. Check the length of the string with the Len function to determine whether Dir found the file.

The following macro demonstrates how to use this function:

```
Sub Macro1()

  If FileExists("C:\Temp\MyNewBook.xlsx") = True Then
    MsgBox "File exists."
  Else
    MsgBox "File does not exist."
  End If

End Sub
```

To implement this macro, you can copy and paste both pieces of code into a standard module.

(See the earlier "Installing Macros" section at the beginning of this chapter.)

Closing All Workbooks at Once

One of the more annoying things in Excel occurs when you try to close many workbooks at once. For each workbook you have opened, you need to reference the workbook, close it, and confirm you want to save changes. There is no easy way to close them all down at one time. This little macro takes care of that annoyance.

This macro uses the Workbooks collection to loop through all the opened workbooks. As the macro loops through each workbook, it saves and closes them.

```
Sub Macro1()

'Step 1: Declare your variables.
  Dim wb As Workbook

'Step 2: Loop through workbooks, save and close.
  For Each wb In Workbooks
    wb.Close SaveChanges:=True
  Next wb

End Sub
```

1. Declares a Workbook object variable. This allows you to enumerate through all the open workbooks, capturing their names as you go.
2. The For Each keywords loops through the open workbooks, saving and closing them. If you don't want to save them, change the SaveChanges argument from True to False.

The best place to store this macro is in your Personal Macro Workbook. (See the earlier "Installing Macros" section at the beginning of this chapter.) This way, the macro is always available to you. The Personal Macro Workbook is loaded whenever you start Excel. In the VBE Project Explorer, it's named personal.xlsb.

Printing All Workbooks in a Directory

If you need to print from multiple workbooks in a directory, you can use this macro.

This code uses the Dir function to iterate through all the .xlsx files in a given directory, capturing each filename. Then you open each file, print it, and close the file.

```
Sub Macro1()

'Step 1: Declare your variables.
  Dim MyFiles As String

'Step 2: Specify a target directory.
  MyFiles = Dir("C:\Temp\*.xlsx")
  Do While Len(MyFiles) > 0

'Step 3: Open Workbooks one by one.
    Workbooks.Open "C:\Temp\" & MyFiles
    ActiveWorkbook.Sheets("Sheet1").PrintOut Copies:=1
    ActiveWorkbook.Close SaveChanges:=False
'Step 4: Next file in the directory.
    MyFiles = Dir
  Loop
```

```
End Sub
```

1. Declares the MyFiles string variable that captures each filename in the enumeration.
2. The Dir function specifies the directory and file type you are looking for. Note that the code here is looking for *.xlsx. The asterisk means that any number of characters can be in the filename as long as the last five are .xlsx. You can use the ? wildcard to specify any single character. For example, Dir(C:\Temp\Budget2021??.xlsx) would find Budget202101.xlsx but not Budget202006.xlsx or Budget2021Jan.xlsx.
3. Opens the file and prints one copy of the first sheet. You may want to change which sheet prints. You can also change the number of copies to print. Because Dir only returns the filename, you have to repeat the path you used from the first time you called Dir in Step 2.
4. When you call Dir with no arguments, it looks for the next file in the directly using the file mask you specified the last time you used Dir. In this case, it's looking for the next file ending in .xlsx. If there are no more files, Dir returns an empty string and the Do While statement exits the loop.

To implement this macro, you can copy and paste it into a standard module. (See the earlier "Installing Macros" section at the beginning of this chapter.)

Preventing the Workbook from Closing Until a Cell Is Populated

There are times when you don't want a user closing a workbook without entering a specific piece of data. In these situations, it would be useful to deny the user the ability to close the workbook until the target cell is filled in (see Figure 4-11). This scenario is where this macro comes in.

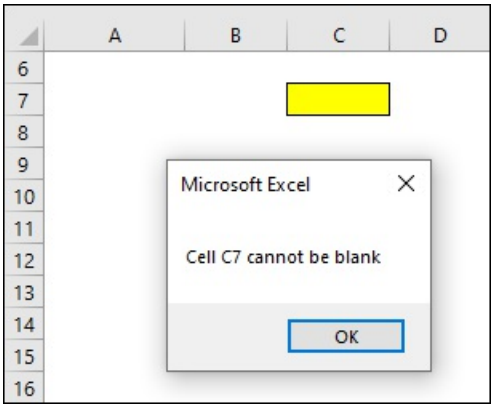


FIGURE 4-11: You can prevent your workbook from closing until a specific cell is populated.

This code is triggered by the workbook's BeforeClose event. When you try to close the workbook, this event fires, running the code within. This macro checks to see whether the target cell (cell C7, in this case) is empty. If it is empty, the close process is cancelled. If C7 is not empty, the workbook saves and closes.

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)

'Step 1: Check to see if Cell C7 is blank.
  If IsEmpty(Sheets("Sheet1").Range("C7").Value) Then
'Step 2: Blank: Cancel the close and tell the user.
    Cancel = True
    MsgBox "Cell C7 cannot be blank"

'Step 3: Not blank; save and close.
  Else
    ActiveWorkbook.Close SaveChanges:=True
  End If

End Sub
```

1. Checks to see whether C7 is blank. The IsEmpty function returns True if the cell is blank.
2. If the cell is blank, sets the Cancel variable to True so Excel doesn't continue with the event's normal action (closing the workbook). Then a message box notifies the user of his stupidity (well, it's not quite that harsh, really).
3. If cell C7 is not blank, the workbook saves and closes.

To implement this macro, you need to copy and paste it into the Workbook_BeforeClose event Code pane, as shown in Figure 4-12. (See the earlier section, "Installing Macros," for

how to do this.) Placing the macro here allows it to run each time you try to close the workbook.

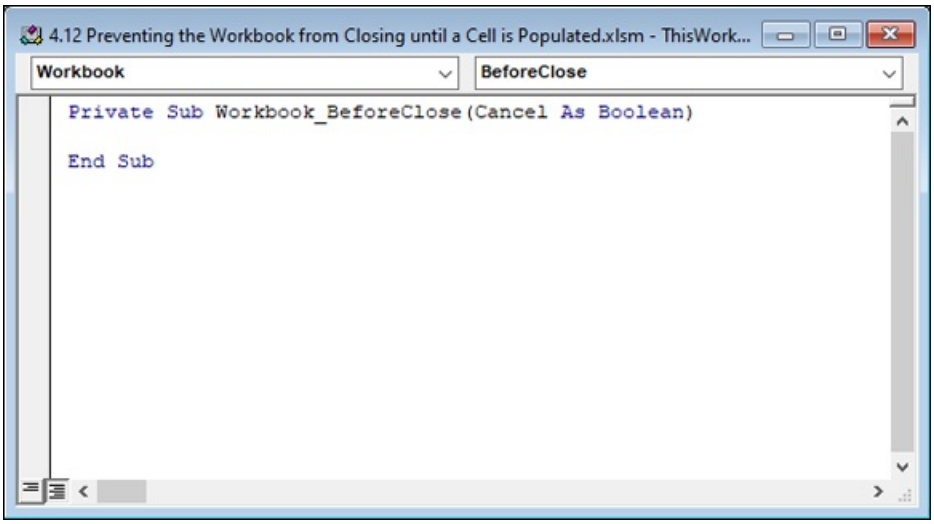


FIGURE 4-12: Type or paste your code in the Workbook_BeforeClose event Code pane.

Creating a Backup of the Current Workbook with Today's Date

Backing up your work is important. Now you can have a macro do it for you. This simple macro saves your workbook to a new file with today's date as part of the name.

The trick to this macro is piecing together the new filename. The new filename has three pieces: the path, the original filename, and today's date.

The path is captured by using the Path property of the ThisWorkbook object. Use the Name property of ThisWorkbook to get the name. Today's date is provided by the Date function.

You're formatting the date (Format(Date, "yyyy-mm-dd")). This is because by default, the Date function returns mm/dd/yyyy. You need to use hyphens rather than forward slashes because the forward slashes would cause the file save to fail. (Windows does not allow forward slashes in filenames.) You also want to have the year first so your files sort properly.

The last piece of the new filename is the original filename. You use the Name property of the ThisWorkbook object to capture that:

```
Sub CreateBackup()

'Step 1: Save workbook with new filename.
  ThisWorkbook.SaveCopyAs _
    Filename:=ThisWorkbook.Path & "\" & _
    Format(Date, "yyyy-mm-dd") & Space(1) & _
    ThisWorkbook.Name

End Sub
```

In the one and only step, the macro builds a new filename and uses the SaveCopyAs method to save the file.

To implement this macro, you can copy and paste it into a standard module. (See the earlier section, "Installing Macros," for more information.)

Chapter 5
Working with Worksheets

IN THIS CHAPTER

- **Adding, naming, copying, and deleting worksheets**
- **Hiding and unhiding, moving, sorting, and grouping worksheets**
- **Protecting and unprotecting all worksheets**
- **Printing and creating a table of contents for your worksheets**
- **Zooming in and out of a worksheet**

Excel analysts often need to automate tasks related to worksheets. Whether it is unhiding all sheets in a workbook, or printing all sheets at the same time, many tasks can be automated to save time and gain efficiencies. This chapter covers some of the more useful macros related to worksheets.

Installing Macros

You can install the macros in this chapter in three places depending on what kind of macro it is and how available you want the macro to be. Event macros are put into the module of the object the event is tied to. Non-event macros go into a standard module and can be in the workbook in which they are meant to run or in a workbook called the Personal Macro Workbook if you want to be able to run them on any workbook.

Event macros

In VBA, objects have events. VBA “listens” for these events to happen and runs special event macros when it detects and event. For example, the workbook object has an Open event. When VBA detects that the workbook has been opened, it looks for a procedure called Workbook_Open in the ThisWorkbook module. If it's there, it runs that macro.

Most of the event macros that you'll use, and most of the examples in the book, are workbook or worksheet events. Workbook event macros go in the ThisWorkbook module and worksheet event macros go in the module for the sheet you want the event code to run on.

To install an event macro, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **In the Project Explorer, find your project/workbook name and click the plus sign next to it to see all the objects.**
3. **Double-click ThisWorkbook.**
4. **Select the event in the right drop-down list.**

5. **Type or paste the code in the module.**

Personal Macro Workbook

There is a special workbook called the Personal Macro Workbook named personal.xlsb. If you record a macro, you can choose this workbook as the destination of the recorded macro. The workbook doesn't exist until you first record a macro into it. After that, it lives in a folder and Excel loads that workbook every time it starts.

The benefit of a workbook that opens every time Excel starts is that all the macros in that workbook are always available to you. If you have a macro that works on the active sheet or whatever range is currently selected, that's a good candidate for the Personal Macro Workbook.

To install a macro in the Personal Macro Workbook, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click personal.xlb in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**



REMEMBER If you don't see personal.xlsb in the Project Explorer, record a macro to the Personal Macro Workbook and Excel will create it for you.

Standard macros

All the other macro examples in this book are installed in a specific workbook in a standard module. If it's not an event macro and it's not a macro you want in the Personal Macro Workbook, then install it using the following steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click the project/workbook name in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**

Adding and Naming a New Worksheet

This chapter starts with one of the simplest worksheet-related automations you can apply with a macro — adding and naming a new worksheet.

If you read through the lines of the code, you'll see this macro is relatively intuitive.

```
Sub AddAndNameWorksheet()

'Step 1: Tell Excel what to do if error.
  On Error GoTo MyError
```

```
'Step 2: Add a sheet and name it.
  Sheets.Add
  ActiveSheet.Name = _
    Format(Now(), "yyyy_mm:dd_hh_mm:ss")
  Exit Sub


'Step 3: If here, an error happened; tell the user.
MyError:
  MsgBox "There is already a sheet called that."

End Sub
```

1. An error occurs if a sheet is given a name that already exists. Excel skips to the line that says MyError (in Step 3) if there is an error.
2. The Add method adds a new sheet. By default, the sheet is called Sheet*xx*, where *xx* represents the number of the sheet. Give the sheet a new name by changing the Name property of the ActiveSheet object. In this case, name the worksheet with the current date and time.

   As with workbooks, each time you add a new sheet via VBA, it automatically becomes the active sheet. If there hasn't been an error yet, exit the procedure so that it skips Step 3 (which should come into play only if an error occurs).
3. Notify the user that the sheet name already exists. Again, this step should only be activated if an error occurs.

To implement this macro, you can copy and paste it into a standard module, as covered in the earlier section, "Installing Macros."

Deleting All but the Active Worksheet

At times, you may want to delete all but the active worksheet. In these situations, you can use this macro.

This macro loops through the worksheets and matches each worksheet name to the active sheet's name. Each time the macro loops, it deletes any worksheet that doesn't match. Note the use of the DisplayAlerts method in Step 4, which turns off Excel's warnings so you don't have to confirm each delete.

```
Sub DeleteAllButActiveSheet()

'Step 1: Declare your variables.
  Dim ws As Worksheet

'Step 2: Loop through all worksheets.
  For Each ws In ThisWorkbook.Worksheets

'Step 3: Check each worksheet name.
    If ws.Name <> ThisWorkbook.ActiveSheet.Name Then

'Step 4: Turn off warnings and delete.
      Application.DisplayAlerts = False
      ws.Delete
      Application.DisplayAlerts = True
    End If
'Step 5: Loop to next worksheet.
  Next ws
```

```
End Sub
```

1. The macro first declares a Worksheet variable called ws. It creates a memory container for each worksheet it loops through.
2. The For Each keywords use the Worksheet control variable to loop through all the worksheets in the workbook. The loop runs once for every worksheet and on each pass the ws variable refers to a different worksheet.

   There is a difference between ThisWorkbook and ActiveWorkbook. The ThisWorkBook object refers to the workbook that the code is contained in. The ActiveWorkBook object refers to the currently active workbook. They can return the same object, but if the workbook running the code is not the active workbook, they return different objects. In this case, you don't want to risk deleting sheets in other workbooks, so you use ThisWorkBook.
3. Compare the active sheet name to the sheet that is currently being looped.
4. If the sheet names are different, delete the sheet. DisplayAlerts stops Excel from asking if you really want to delete the sheet. If you want to be warned before deleting each sheet, you can omit the Application.DisplayAlerts = False. This ensures you get the message in Figure 5-1, allowing you to back out of the decision to delete worksheets.
5. The Next keyword tells Excel it's the end of the loop and to go back to the start if there are any worksheets yet to be evaluated. After all sheets are evaluated, the macro ends.

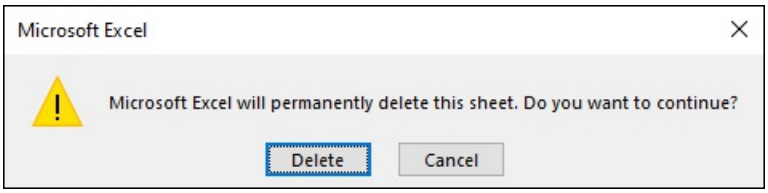


FIGURE 5-1: Omit the Application.DisplayAlerts = False line in the macro to ensure you get the opportunity to cancel the deletion.

To implement this macro, you can copy and paste it into a standard module. See the earlier section, "Installing Macros," for more information.



REMEMBER Note that when you use ThisWorkbook in a macro rather than ActiveWorkbook, you can't run the macro from the Personal Macro Workbook. This is because ThisWorkbook refers to the Personal Macro Workbook, not the workbook to which the macro should apply.

Hiding All but the Active Worksheet

You may not want to delete all but the active sheet as you did in the last macro. Instead, a more gentle option is to simply hide the sheets. Excel doesn't let you hide all sheets in a workbook; at least one has to be showing. However, you can hide all but the active sheet.

This macro loops the worksheets and matches each worksheet name to the active sheet's name. Each time the macro loops, it hides any unmatched worksheet.

```
Sub HideAllButActiveSheet()

'Step 1: Declare your variables.
  Dim ws As Worksheet

'Step 2: Start looping through all worksheets.
  For Each ws In ThisWorkbook.Worksheets

'Step 3: Check each worksheet name.
    If ws.Name <> ThisWorkbook.ActiveSheet.Name Then

'Step 4: Hide the sheet.
      ws.Visible = xlSheetHidden
    End If

'Step 5: Loop to next worksheet.
  Next ws

End Sub
```

1. Declares a Worksheet variable called ws. This creates a memory container for each worksheet the macro loops through.
2. The For Each keywords use the Worksheet control variable to loop through all the worksheets in the workbook. The loop runs once for every worksheet and on each pass the ws variable refers to a different worksheet.

   REMEMBER There is a difference between ThisWorkbook and ActiveWorkbook. The ThisWorkBook object refers to the workbook that the code is contained in. The ActiveWorkBook object refers to the currently active workbook. They can return the same object, but if the workbook running the code is not the active workbook, they return different objects. In this case, you don't want to risk deleting sheets in other workbooks, so be sure to use ThisWorkBook.
3. Compares the active sheet name to the sheet the loop is on.
4. If the sheet names are different, the sheet is hidden.
5. The Next keyword tells Excel it's the end of the loop and to go back to the start if there are any worksheets yet to be evaluated. After all the sheets are evaluated, the macro ends.

   TIP This example uses the built-in constant xlSheetHidden in the macro. This is the same as right-clicking a sheet and choosing Hide. Using this method means a user can right-click any tab and choose Unhide. But there is another, more clandestine, hidden state than the default. If you use xlSheetVeryHidden to hide your sheets, users won't be able to see them at all — not even if they right-click any tab and choose Unhide. The only way to unhide a sheet hidden in this manner is to use VBA.

To implement this macro, you can copy and paste it into a standard module, as outlined in the

first section in this chapter, "Installing Macros."

Unhiding All Worksheets in a Workbook

You can unhide all hidden sheets by right-clicking a visible sheet and choosing Unhide to show the Unhide dialog box shown in Figure 5-2.

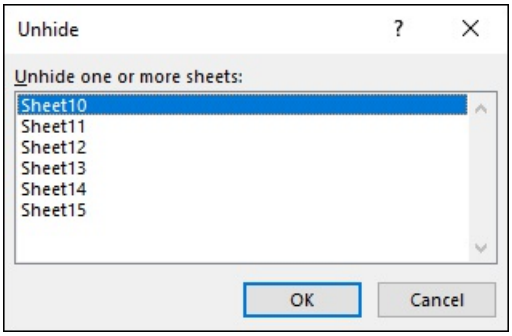


FIGURE 5-2: Without a macro, you're stuck using Excel's Unhide dialog box to unhide one worksheet at a time.

You can use the Shift and Ctrl keys to select multiple workbooks to unhide. If you want to save yourself some clicks, you can use this macro.

This macro loops the worksheets and makes sure each worksheet is visible.

```
Sub UnhideAllSheets()

'Step 1: Declare your variables.
  Dim ws As Worksheet

'Step 2: Start looping through all worksheets.
  For Each ws In ActiveWorkbook.Worksheets
'Step 3: Loop to next worksheet.
      ws.Visible = xlSheetVisible
  Next ws

End Sub
```

1. Declares a Worksheet variable called ws. This creates a memory container for each worksheet the macro loops through.
2. Uses For Each to tell Excel to enumerate through all worksheets in this workbook.
3. Changes the Visible property to xlSheetVisible. Then loops to get the next worksheet.

The best place to store this macro is in your Personal Macro Workbook, as outlined in the first section of this chapter, "Installing Macros." That way, the macro is always available to you.

Moving Worksheets Around

You've probably had to rearrange your workbook so that some sheets came before or after other sheets. If you find that you often have to do this, here is a macro that can help.

When you want to rearrange sheets, you use the Move method of either the Sheets object or

the ActiveSheet object. When using the Move method, you need to specify where to move the sheet to. You can do this using the After argument, the Before argument, or you can omit both arguments. If you omit both, the worksheet is moved to a new workbook.

```
SSub MoveSheets()

'Move the active sheet to the end.
  ActiveSheet.Move After:=Worksheets(Worksheets.Count)

'Move the active sheet to the beginning.
  ActiveSheet.Move Before:=Worksheets(1)

'Move Sheet 1 before Sheet 12.
  Sheets("Sheet1").Move Before:=Sheets("Sheet12")

End Sub
```

This macro demonstrates how to move the active worksheet to three locations.

- **Move the active sheet to the end:** When you need to move a worksheet to the end of the workbook, Excel moves the sheet after the last sheet. There is no code in VBA that lets you point to "the last sheet." But you *can* find the count of worksheets, and then use that number as an index for the Worksheets object. This means that you can enter something such as Worksheets(1) to point to the first sheet in a workbook. You can enter Worksheet(3) to point to the third sheet in the workbook. To point to the last sheet in the workbook, you can replace the index number with the Worksheets.Count property. Worksheets.Count gives you the total number of worksheets, which is the same number as the index for the last sheet. Thus Worksheet(Worksheets.Count) points to the last sheet.
- **Move the active sheet to the beginning:** Moving a sheet to the beginning of the workbook is simple. You use Worksheets(1) to point to the first sheet in the workbook, and then move the active sheet before that one.
- **Move Sheet 1 before Sheet *X*:** You can also move a sheet before or after another sheet simply by calling that other sheet by name. In the example demonstrated in the previous macro, Sheet1 is moved before Sheet12.

The best place to store this kind of macro is in your Personal Macro Workbook, as outlined in the first section of this chapter, "Installing Macros." This way, the macro is always available to you.

Sorting Worksheets by Name

You may often need to sort worksheets alphabetically by name (see Figure 5-3). You would think Excel would have a native function to do this, but alas, it does not. If you don't want to manually sort your spreadsheets anymore, you can use this macro to do it for you.

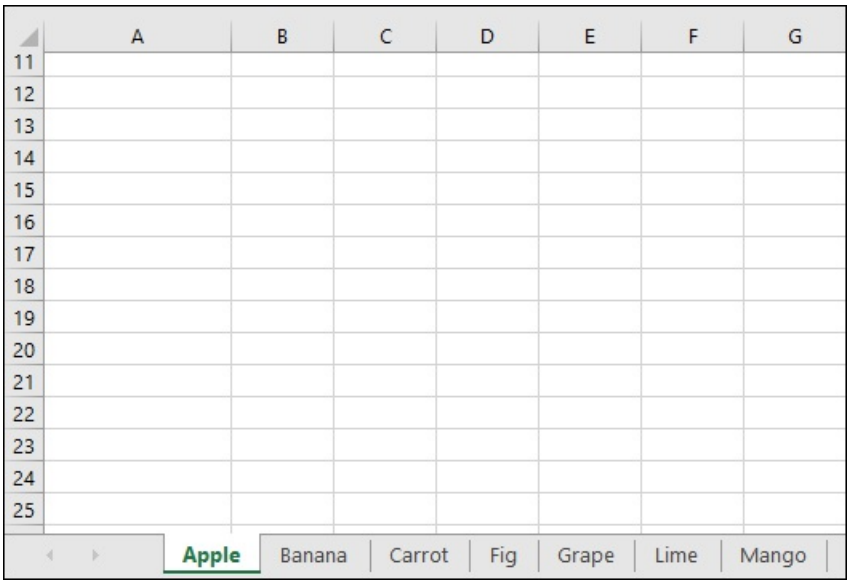


FIGURE 5-3: It's often useful to have your worksheets sorted in alphabetical order.

This macro looks more complicated than it is. The activity in this macro is actually fairly simple. It iterates through the sheets in the workbook, comparing the current sheet to the previous one. If the name of the previous sheet is greater than the current sheet (alphabetically), the macro moves the current sheet before it. By the time all the iterations are done, you've got a sorted workbook!

```
Sub SortByName()

'Step 1: Declare your variables.
  Dim idxCurr As Long
  Dim idxPrev As Long

'Step 2: Set the starting counts and start looping.
  For idxCurr = 2 To Sheets.Count
    For idxPrev = 1 To idxCurr - 1

'Step 3: Check current sheet against previous sheet.
      If UCase(Sheets(idxPrev).Name) > _
        UCase(Sheets(idxCurr).Name) Then

'Step 4: If move current sheet before previous.
        Sheets(idxCurr).Move _
          Before:=Sheets(idxPrev)
      End If

'Step 5: Loop back around to iterate again
    Next idxPrev
  Next idxCurr

End Sub
```



REMEMBER Note this technique is doing a text-based sorting. So you may not get the results you were expecting when working with number-based sheet names. For example, Sheet10 comes before Sheet2 because textually, 1 comes before 2. Excel does not do the numbers-based sorting that says 2 comes before 10.

1. Declares two Long variables. The idxCurr variable holds the index number for the current

sheet iteration, and the idxPrev variable holds the index number for the previous sheet iteration.

2. Creates an outer loop using idxCurr and an inner loop using idxPrev. The outer loop starts at 2 so you don't compare the first sheet to itself and the inner loop stops one before the end so you don't compare the last sheet to itself.
3. Checks to see whether the name of the previous sheet is greater than that of the current sheet.



TIP Use the UCase function to get both names in uppercase. This prevents sorting errors due to differing cases.

4. If the previous sheet name is greater than the current sheet name, the Move method moves the current sheet before the previous sheet.
5. Goes back to the start of the loop. Every iteration of the loop increments both variables up one number until the last worksheet is touched. After all iterations have been spent, the macro ends.

The best place to store this macro is in your Personal Macro Workbook, as outlined in the first section of this chapter, "Installing Macros." This way, the macro is always available to you.

Grouping Worksheets by Color

This macro is useful if you assign colors to your worksheet tabs. You can right-click any tab and select the Tab Color option (shown in Figure 5-4) to choose a color for your tab.

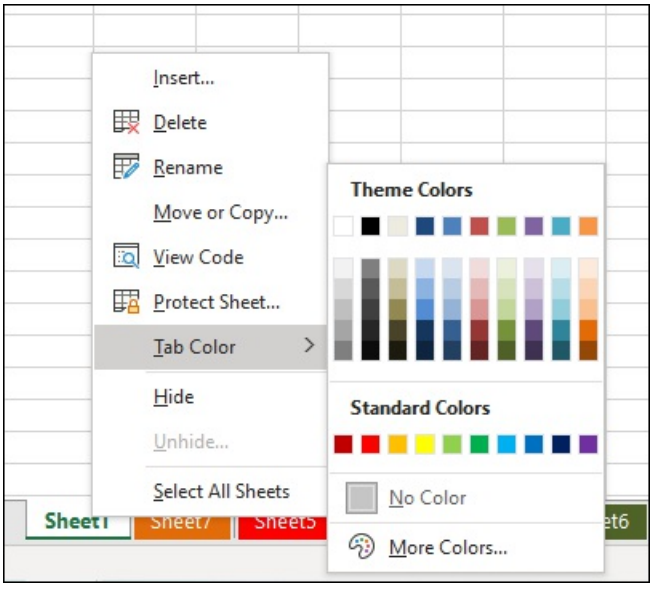


FIGURE 5-4: You can right-click any worksheet to choose a tab color for the sheet.

This allows for visual confirmation that the data in a certain tab is somehow related to another tab because both have the same color. When you have many colored sheets, it’s often useful to group tabs of similar color together for ease of navigation.

This macro groups worksheets based on their tab colors. You may think it's impossible to sort or group by color, but Excel offers a way. Excel assigns an index number to every color. A light yellow color may have an index number of 36, whereas a maroon color has the index number 42.

This macro iterates through the sheets in the workbook, comparing the tab color index of the current sheet to that of the previous one. If the previous sheet has the same color index number as the current sheet, the macro moves the current sheet before it. By the time all the iterations are done, all the sheets are grouped together based on their tab colors.

```
Sub GroupByColor()

'Step 1: Declare your variables.
  Dim idxCurr As Long
  Dim idxPrev As Long

'Step 2: Set the starting counts and start looping.
  For idxCurr = 2 To Sheets.Count
    For idxPrev = 1 To idxCurr - 1

'Step 3: Check current sheet against previous sheet.
      If Sheets(idxPrev).Tab.ColorIndex = _
        Sheets(idxCurr).Tab.ColorIndex Then

'Step 4: Move current sheet before previous.
        Sheets(idxPrev).Move _
          Before:=Sheets(idxCurr)
      End If

'Step 5: Loop back around to iterate again.
    Next idxPrev
  Next idxCurr

End Sub
```

1. Declares two Long variables. The idxCurr variable holds the index number for the current sheet iteration, and the idxPrev variable holds the index number for the previous sheet iteration.
2. Creates an outer loop using idxCurr and an inner loop using idxPrev. The outer loop starts at 2 so you don't compare the first sheet to itself and the inner loop stops one before the end so you don't compare the last sheet to itself.
3. Checks to see whether the color index of the previous sheet is the same as that of the current sheet using the Tab.ColorIndex property.
4. If the color index of the previous sheet is equal to the color index of the current sheet, the Move method moves the current sheet before the previous sheet.
5. Goes back to the start of the loop. Every iteration of the loop increments both variables up one number until the last worksheet is touched. After all iterations have been spent, the macro ends.

The best place to store this macro is in your Personal Macro Workbook, as outlined in the first section of this chapter, “Installing Macros.” This way, the macro is always available to

you.

Copying a Worksheet to a New Workbook

In Excel, you can manually copy an entire sheet to a new workbook by right-clicking the target sheet and selecting the Move or Copy option. Unfortunately, if you try to record a macro while you do this, the macro recorder fails to accurately write the code to reflect the task. So if you need to programmatically copy an entire sheet to a brand new workbook, this macro delivers.

In this macro, the active sheet is first being copied. Then you use the Before parameter to send the copy to a new workbook that is created on the fly. The copied sheet is positioned as the first sheet in the new workbook.

The use of the ThisWorkbook object is important here. This ensures that the active sheet being copied is from the workbook that the code is in, not the newly created workbook.

```
Sub CopySheetToNewWb()

'Copy sheet, and send to new workbook.
  ThisWorkbook.ActiveSheet.Copy _
    Before:=Workbooks.Add.Worksheets(1)

End Sub
```

To implement this macro, you can copy and paste it into a standard module. See the first section in this chapter, “Installing Macros,” for more information.

Creating a New Workbook for Each Worksheet

Many Excel analysts need to parse their workbooks into separate books per worksheet tab. In other words, they need to create a new workbook for each of the worksheets in their existing workbook. You can imagine what an ordeal this would be if you were to do it manually. The following macro helps automate that task.

In this macro, you are looping the worksheets, copying each sheet, and then sending the copy to a new workbook that is created on the fly. The thing to note here is that the newly created workbooks are being saved in the same directory as your original workbook, with the same filename as the copied sheet (wb.SaveAs ThisWorkbook.Path & "\" & ws.Name).

```
Sub NewWbForEachSheet()

'Step 1: Declare all the variables.
  Dim ws As Worksheet
  Dim wb As Workbook

'Step 2: Start the looping through sheets.
  For Each ws In ThisWorkbook.Worksheets

'Step 3: Create new workbook and save it.
    Set wb = Workbooks.Add
    wb.SaveAs ThisWorkbook.Path & "\" & ws.Name

'Step 4: Copy the target sheet to the new workbook.
    ws.Copy Before:=wb.Worksheets(1)
    wb.Close SaveChanges:=True

'Step 5: Loop back around to the next worksheet.
  Next ws

End Sub
```



REMEMBER Not all valid worksheet names translate to valid filenames.

Windows has specific rules that prevent you from naming files with certain characters. You cannot use these characters when naming a file: back slash (\), forward slash (/), colon (:), asterisk (*), question mark (?), pipe (|), double quote (“), greater than (>), and less than (<).

The twist is that you can use a few of these restricted characters in your sheet names; specifically, double quote (“), pipe (|), greater than (>), and less than (<).

So as you’re running this macro, naming the newly created files to match the sheet name may cause an error. For example, the macro throws an error when creating a new file from a sheet called “May| Revenue” (because of the pipe character).

Long story short, avoid naming your worksheets with the restricted characters just mentioned.

1. Declares two object variables. The ws variable creates a memory container for each worksheet the macro loops through. The wb variable creates the container for the new workbooks you create.
2. Loops through the sheets. The use of the ThisWorkbook object ensures that the active

sheet being copied is from the workbook the code is in, not the newly created workbook.

3. Adds a new workbook and save it. You save this new book in the same path as the original workbook (ThisWorkbook). The filename is set to be the same name as the currently active sheet.
4. Copies the currently active sheet, using the Before parameter to send it to the new book as the first sheet. Then saves and closes the newly created workbook.
5. Loops back to get the next sheet. After all of the sheets are evaluated, the macro ends.

To implement this macro, you can copy and paste it into a standard module. See the earlier section, "Installing Macros," to find out how.

Printing Specified Worksheets

If you want to print specific sheets manually in Excel, you need to hold down the Ctrl key on the keyboard, select the sheets you want to print, and then click Print. If you do this often enough, you may consider using this very simple macro.

This one is easy. All you have to do is pass the sheets you want printed in an array as seen here in this macro, and then use the PrintOut method to trigger the print job. All the sheets you have entered are printed in one go.

```
Sub PrintSheets()

'Print certain sheets.
  ActiveWorkbook.Sheets( _
    Array("Sheet1", "Sheet3", "Sheet5")).PrintOut Copies:=1

End Sub
```

Want to print all worksheets in a workbook? This one is even easier.

```
Sub PrintAllSheets()

'Print all sheets.
  ActiveWorkbook.Worksheets.PrintOut Copies:=1

End Sub
```

The best place to store this macro is in your Personal Macro Workbook, as outlined in the first section of this chapter, "Installing Macros." This way, the macro is always available to you.

Protecting All Worksheets

Before you distribute your workbook, you may want to apply sheet protection to all the sheets. However, as shown in Figure 5-5, Excel disables the Protect Sheet command if you try to protect multiple sheets at one time. You're forced to protect one sheet at a time. You can use this macro to save you from protecting each sheet manually.

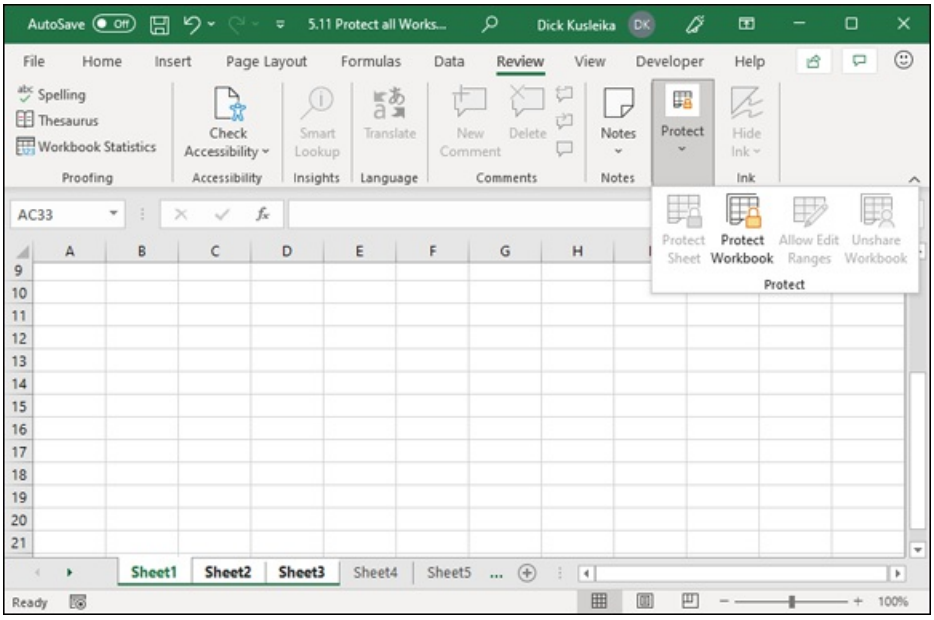


FIGURE 5-5: The Protect Sheet command is disabled if you try to protect more than one sheet at a time.

In this macro, you're looping the worksheets and simply applying protection with a password. The Password argument defines the password needed to remove the protection. The Password argument is completely optional. If you omit it altogether, the sheet is still protected; you just won't need to enter a password to unprotect it. Also, be aware that Excel passwords are case-sensitive, so you want pay attention to the exact capitalization you are using.

```
Sub ProtectAllSheets()

'Step 1: Declare your variables.
  Dim ws As Worksheet

'Step 2: Start looping through all worksheets.
  For Each ws In ActiveWorkbook.Worksheets

'Step 3: Protect and loop to next worksheet.
    ws.Protect Password:="RED"
  Next ws

End Sub
```

1. Declares a Worksheet variable called ws. This creates a memory container for each worksheet you loop through.
2. Starts the looping, telling Excel you want to enumerate through all worksheets in this workbook.
3. Applies protection with the given password, and then loops back to get the next worksheet.

The best place to store this macro is in your Personal Macro Workbook, as outlined in the first section of this chapter, "Installing Macros." This way, the macro is always available to you.

Unprotecting All Worksheets

You may find yourself constantly having to unprotect multiple worksheets manually. However, as shown in Figure 5-6, Excel disables the Unprotect Sheet command if you try to unprotect multiple sheets at one time. You're forced to unprotect one sheet at a time. You can use this macro to save you from unprotecting each sheet manually.

This macro loops the worksheets and uses the Password argument to unprotect each sheet.

```
Sub UnprotectAllSheets()

'Step 1: Declare your variables.
  Dim ws As Worksheet

'Step 2: Start looping through all worksheets.
  For Each ws In ActiveWorkbook.Worksheets

'Step 3: Loop to next worksheet.
    ws.Unprotect Password:="RED"
  Next ws

End Sub
```

1. Declares an Worksheet variable called ws. This creates a memory container for each worksheet you loop through.
2. Starts the looping, telling Excel to enumerate through all worksheets in this workbook.
3. Unprotects the active sheet, providing the password as needed, and then loops back to get the worksheet.

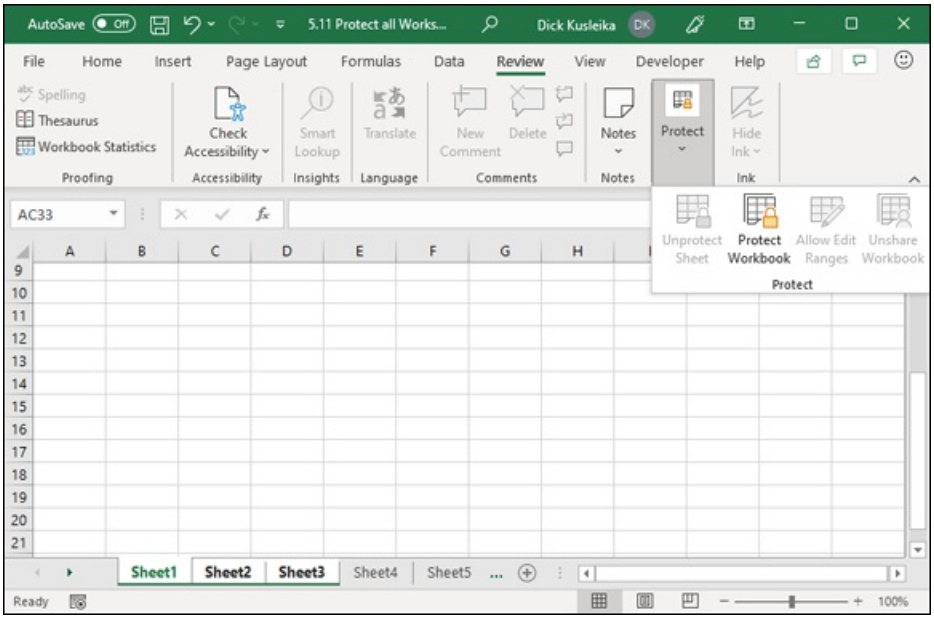


FIGURE 5-6: The Unprotect Sheet command is disabled if you try to unprotect more than one sheet at a time.

Obviously, the assumption is that all the worksheets that need to be unprotected have the same password. If this is not the case, you need to explicitly unprotect each sheet with its corresponding password.

```
Sub UnprotectAllSheets2()

  Sheets("Sheet1").Unprotect Password:="RED"
```

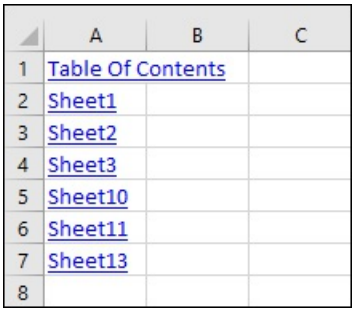
```
  Sheets("Sheet2").Unprotect Password:="BLUE"
  Sheets("Sheet3").Unprotect Password:="YELLOW"
  Sheets("Sheet4").Unprotect Password:="GREEN"

End Sub
```

The best place to store this kind of a macro is in your Personal Macro Workbook, as outlined in the first section of this chapter, "Installing Macros." This way, the macro is always available to you.

Creating a Table of Contents for Your Worksheets

Outside of sorting worksheets, creating a table of contents for the worksheets in a workbook is the most commonly requested Excel macro. The reason is probably not lost on you if you often work with files that have more worksheet tabs than can easily be seen or navigated. A table of contents such as the one shown in Figure 5-7 definitely helps.



	A	B	C
1	Table Of Contents		
2	Sheet1		
3	Sheet2		
4	Sheet3		
5	Sheet10		
6	Sheet11		
7	Sheet13		
8			

FIGURE 5-7: A table of contents can help you more easily navigate your workbook.

The following macro not only creates a list of worksheet names in the workbook, but it also adds hyperlinks so that you can easily jump to a sheet with a simple click.

It's easy to get intimidated when looking at this macro. A lot is going on here. However, if you step back and consider the few simple actions it does, it becomes a little less scary:

- It removes any previous Table of Contents sheet.
- It creates a new Table of Contents sheet.
- It grabs the name of each worksheet and pastes it on the Table of Contents.
- It adds a hyperlink to each entry in the Table of Contents.

That doesn't sound so bad. Now look at the code:

```
Sub CreateTOC()

'Step 1: Declare variables.
  Dim i As Long

'Step 2: Delete previous TOC if exists.
  On Error Resume Next
    Application.DisplayAlerts = False
    Sheets("Table Of Contents").Delete
    Application.DisplayAlerts = True
  On Error GoTo 0
```

```
'Step 3: Add a new TOC sheet as the first sheet.
  ThisWorkbook.Sheets.Add _
    Before:=ThisWorkbook.Worksheets(1)
  ActiveSheet.Name = "Table Of Contents"

'Step 4: Start the i counter.
  For i = 1 To Sheets.Count

'Step 5: Select next available row.
    ActiveSheet.Cells(i, 1).Select

'Step 6: Add sheet name and hyperlink.
    ActiveSheet.Hyperlinks.Add _
      Anchor:=ActiveSheet.Cells(i, 1), _
      Address:="", _
      SubAddress:="'" & Sheets(i).Name & "'!A1", _
      TextToDisplay:=Sheets(i).Name

'Step 7: Loop back increment i.
  Next i

End Sub
```

1. Declares a Long integer variable called i to serve as the counter as the macro iterates through the sheets.

   REMEMBER Note that this macro is not looping through the sheets the way previous macros in this chapter did. In previous macros, you looped through the worksheets collection and selected each worksheet there. This macro uses a counter (your i variable). The main reason is because you not only have to keep track of the sheets, but you also have to manage to enter each sheet name on a new row into a table of contents. The idea is that as the counter progresses through the sheets, it also serves to move the cursor down in the Table of Contents so each new entry goes on a new row.
2. Attempts to delete any previous sheet called Table of Contents. Because there may not be any Table of Contents sheet to delete, you have to start with the On Error Resume Next error statement. This tells Excel to continue the macro if an error is encountered here. You then set DisplayAlerts to False so you don't have to confirm the deletion. Delete the sheet and reset DisplayAlerts and the error handler to trap all errors again by entering On Error GoTo 0.
3. Adds a new sheet to the workbook using the Before argument to position the new sheet as first sheet. Name the sheet Table of Contents. Remember that when you add a new worksheet, it automatically becomes the active sheet. Because this new sheet has the focus throughout the macro, any references to ActiveSheet in this code refer to the Table of Contents sheet.
4. Starts the i counter at 1 and ends it at the count of all sheets in the workbook. Again, instead of looping through the Worksheets collection like in previous macros, the i counter stores an index number that is passed to the Sheets object. When the maximum number is reached, the macro ends.
5. Selects the corresponding row in the Table of Contents sheet. That is to say, if the i counter is on 1, it selects the first row in the Table of Contents sheet. If the i counter is at

2, it selects the second row, and so on.

You are able to do this using the Cells property. The Cells property provides an extremely handy way of selecting ranges through code. It requires only relative row and column positions as parameters. So Cells(1,1) translates to row 1, column 1 (or cell A1). Cells(5, 3) translates to row 5, column 3 (or cell C5).



TIP The numeric parameters in the Cells item are particularly handy when you want to loop through a series of rows or columns using an incrementing index number.

6. Adds a hyperlink using the Hyperlinks.Add method to add the sheet name and hyperlinks to the selected cell. The Anchor parameter determines which cell the hyperlink is in. The Address is blank because all the links are within the workbook that contains the hyperlink. To link outside of the workbook, set the Address property to the other file. The SubAddress is the sheet name and cell address within the file. The TextToDisplay argument is what shows in the hyperlink cell.
7. Loops back to increment the i counter to the next count. When the i counter reaches a number that equals the count of worksheets in the workbook, the macro ends.

To implement this macro, you can copy and paste it into a standard module. See the section at the beginning of this chapter, "Installing Macros," for details.

Zooming In and Out of a Worksheet with Double-Click

Some spreadsheets are huge. Sometimes, you're forced to shrink the font size down so that you can see a decent portion of the spreadsheet onscreen. If you find that you're constantly zooming in and out of a spreadsheet, alternating between scanning large sections of data and reading specific cells, here is a handy macro that auto-zooms on double-click.

With this macro in place, you can double-click a cell in the spreadsheet to zoom in 200 percent. Double-click again and Excel zooms out to 100 percent. Obviously, you can change the values and complexity in the code to fit your needs.

```
Private Sub Worksheet_BeforeDoubleClick(ByVal Target As Range, _
  Cancel As Boolean)

'Check current zoom state
'Zoom to 100% if not at 100
'Zoom 200% if currently at 100
  If ActiveWindow.Zoom <> 100 Then
    ActiveWindow.Zoom = 100
  Else
    ActiveWindow.Zoom = 200
  End If

  Cancel = True
End Sub
```



TIP The default behavior of double-clicking a cell is that it goes into edit mode. This macro sets the Cancel argument to True to prevent that behavior.

To implement this macro, you need to copy and paste it into the Worksheet_BeforeDoubleClick event Code pane, as shown in Figure 5-8. See the first section at the beginning of this chapter, "Installing Macros," for more information. Placing the macro there allows it to run each time you double-click the sheet.

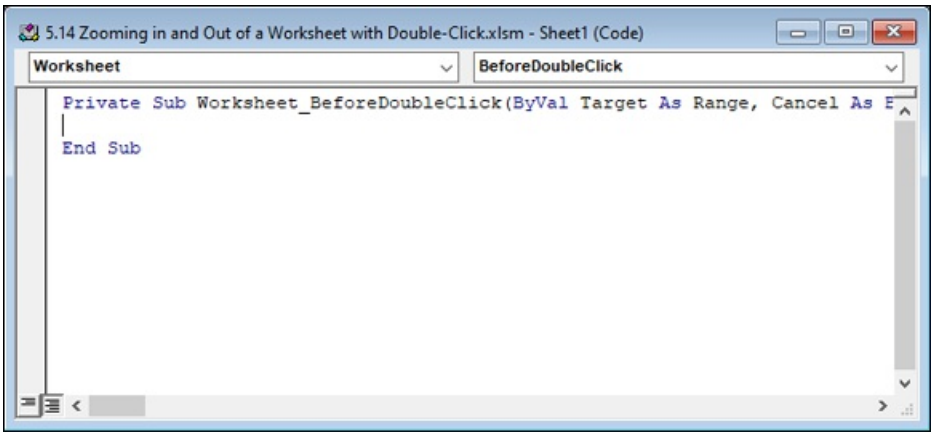


FIGURE 5-8: Type or paste your code into the Worksheet_BeforeDoubleClick event code window.

Highlighting the Active Row and Column

When looking at a table of numbers, it would be nice if Excel automatically highlighted the row and column you're on (as shown in Figure 5-9). This effect gives your eyes a lead line up and down the column as well as left and right across the row.

	A	B	C	D	E	F	G	H
7								
8		Jan	Feb	Mar	Apr	May	Jun	Jul
9	Product 1	$74,084	$41,353	$37,032	$77,941	$35,221	$55,695	$4,593
10	Product 2	$70,049	$42,425	$51,966	$25,159	$35,929	$16,817	$53,739
11	Product 3	$13,513	$98,468	$18,818	$27,001	$11,373	$2,347	$69,236
12	Product 4	$72,705	$25,553	$68,709	$86,278	$58,278	$47,659	$29,373
13	Product 5	$35,637	$81,467	$83,445	$51,797	$58,971	$52,776	$78,204
14	Product 6	$61,118	$71,932	$42,153	$20,370	$44,917	$25,386	$44,196
15	Product 7	$42,303	$19,757	$78,250	$32,396	$1,863	$30,091	$80,849
16	Product 8	$74,735	$53,599	$52,357	$55,778	$89,745	$43,096	$63,467
17	Product 9	$29,764	$31,476	$92,661	$76,510	$93,957	$90,794	$53,193
18	Product 10	$37,577	$68,726	$42,900	$60,592	$7,627	$17,429	$80,228
19	Product 11	$98,304	$19,809	$56,834	$62,311	$54,039	$55,888	$61,837
20	Product 12	$64,827	$85,195	$16,953	$47,824	$26,565	$52,672	$4,760
21								

FIGURE 5-9: A highlighted row and column makes it easy to track data horizontally and vertically.

The following macro enables the effect shown in Figure 5-9 with just a simple double-click.

When the macro is in place, Excel highlights the row and column for the active cell, greatly improving your ability to view and edit a large grid.

Take a look at how this macro works:

```
Private Sub Worksheet_BeforeDoubleClick(ByVal Target As Range, Cancel As Boolean)

  Application.Union(Target.EntireColumn, _
    Target.EntireRow).Select

End Sub
```

The Union method takes multiple ranges of cells and makes them a single range of cells. In this example, it takes the whole column and the whole row and creates one range containing both. Then it's a simple call to the Select method to highlight the range.

To implement this macro, you need to copy and paste it into the Worksheet_BeforeDoubleClick event Code pane. See the first section in this chapter, "Installing Macros," for more information. Placing the macro there allows it to run each time you double-click the sheet.

Part 3

One-Touch Data Manipulation

IN THIS PART …

Go beyond basic macros and look at some advanced techniques for navigating ranges through VBA.

See how macros can automate the selection and manipulation of specific cells.

Explore how you can use macros to clean and transform the data in your workbooks.

Uncover techniques that can help you automate the export of your Excel data.

Chapter 6

Feeling at Home on the Range

IN THIS CHAPTER

- **Selecting and formatting a range**
- **Creating and selecting a named range**
- **Looping through a range of cells**
- **Inserting blank rows in a range**
- **Unhiding all rows and columns**
- **Deleting blank rows**
- **Deleting blank columns**
- **Limiting range movement to a particular area**
- **Selecting and formatting all formulas in a workbook**
- **Finding and selecting the first blank row or column**

One of the most frequent things you do in Excel is navigate the worksheet. When you work with Excel manually, you're constantly navigating to appropriate ranges, finding the last row, moving to the last column, hiding and unhiding ranges, and so on. This all comes instinctively as part of doing work in Excel.

When you attempt to automate your work through VBA, navigating your spreadsheet remains an important part of the automation process. In many cases, you need to dynamically navigate and manipulate Excel ranges, just as you would manually — only through VBA code. This chapter provides some of the most commonly used macros in terms of navigating and working with ranges.

Installing Macros

You can install the macros in this chapter in three places depending on what kind of macro it is and how available you want the macro to be. Event macros are put into the module of the object the event is tied to. Non-event macros go into a standard module and can be in the workbook in which they are meant to run or in a workbook called the Personal Macro Workbook if you want to be able to run them on any workbook.

Event macros

In VBA, objects have events. VBA "listens" for these events to happen and runs special event macros when it detects and event. For example, the workbook object has an Open event. When VBA detects that the workbook has been opened, it looks for a procedure called Workbook_Open in the ThisWorkbook module. If it's there, it runs that macro.

Most of the event macros that you'll use, and most of the examples in the book, are workbook or worksheet events. Workbook event macros go in the ThisWorkbook module and worksheet event macros go in the module for the sheet you want the event code to run on.

To install an event macro, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **In the Project Explorer, find your project/workbook name and click the plus sign next to it to see all the objects.**
3. **Double-click ThisWorkbook.**
4. **Select the event in the right drop-down list.**
5. **Type or paste the code in the module.**

Personal Macro Workbook

There is a special workbook called the Personal Macro Workbook named personal.xlsb. If you record a macro, you can choose this workbook as the destination of the recorded macro. The workbook doesn't exist until you first record a macro into it. After that, it lives in a folder and Excel loads that workbook every time it starts.

The benefit of a workbook that opens every time Excel starts is that all the macros in that workbook are always available to you. If you have a macro that works on the active sheet or whatever range is currently selected, that's a good candidate for the Personal Macro Workbook.

To install a macro in the Personal Macro Workbook, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click personal.xlb in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**



REMEMBER If you don't see personal.xlsb in the Project Explorer, record a macro to the Personal Macro Workbook and Excel will create it for you.

Standard macros

All the other macro examples in this book are installed in a specific workbook in a standard module. If it's not an event macro and it's not a macro you want in the Personal Macro Workbook, then install it using the following steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click the project/workbook name in the Project Explorer.**

3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**

Selecting and Formatting a Range

One of the basic things you need to do in VBA is select a specific range to do something with it. This simple macro selects the range D5:D16.

In this macro, you explicitly define the range to select by using the Range object.

```
Sub Macro1()

  Range("D5:D16").Select

End Sub
```

After the range of cells is selected, you can use any of the Range properties to manipulate the cells. Alter this macro so that the range is colored yellow, converted to number formatting, and bold.

```
Sub Macro1()

  Range("D5:D16").Select
  Selection.NumberFormat = "#,##0"
  Selection.Font.Bold = True
  Selection.Interior.Color = vbYellow

End Sub
```



TIP You don’t have to memorize all the properties of the cell object to manipulate them. You can simply record a macro, do your formatting, and then look at the code that Excel has written. After you've seen what the correct syntax is, you can apply it as needed. Many Excel programmers start learning VBA this way!

Selection is referred to many times in the previous sample code. To write more efficient code, you can simply refer to the range, using the With…End With statement. This statement tells Excel that any action you perform applies to the object to which you’ve pointed. Note that this macro doesn’t actually select the range at all. This is a key point. In a macro, you can work with a range without selecting it first.

```
Sub Macro1()

  With Range("D5:D16")
    .NumberFormat = "#,##0"
    .Font.Bold = True
    .Interior.Color = vbYellow
  End With

End Sub
```



TIP Another way you can select a range is by using the Cells property.

The Cells property gives you an extremely handy way of selecting ranges through code. It

requires only row and column positions as parameters. Cells (5,4) translates to row 5, column 4 (or Cell D5). Cells (16, 4) translates to row 16, column 4 (or cell D16).

If you want to select a range of cells, simply pass two items into the Range object. This macro performs the same selection of range D5:D16:

```
Sub Macro1()

  Range(Cells(5, 4), Cells(16, 4)).Select

End Sub
```

Here is the full formatting code using the Cells property. Again, note that this macro doesn't actually select the range you're altering at all. You can work with a range without selecting it first.

```
Sub Macro1()

  With Range(Cells(5, 4), Cells(16, 4))
    .NumberFormat = "#,##0"
    .Font.Bold = True
    .Interior.Color = vbYellow
  End With

End Sub
```

To implement this kind of a macro, you can copy and paste it into a standard module. See the earlier section in this chapter, "Installing Macros," for the details.

Creating and Selecting Named Ranges

One of the more useful features in Excel is the ability to name your range (that is, to give your range a user-friendly name, so that you can more easily identify and refer to it via VBA).

Here are the steps you would perform to create a named range manually:

1. **Select the range you want to name.**
2. **On the Ribbon's Formulas tab, click the Define Name command (see Figure 6-1).**
3. **Give the chosen range a user-friendly name in the New Name dialog box, as shown in Figure 6-2.**
4. **Click OK.**

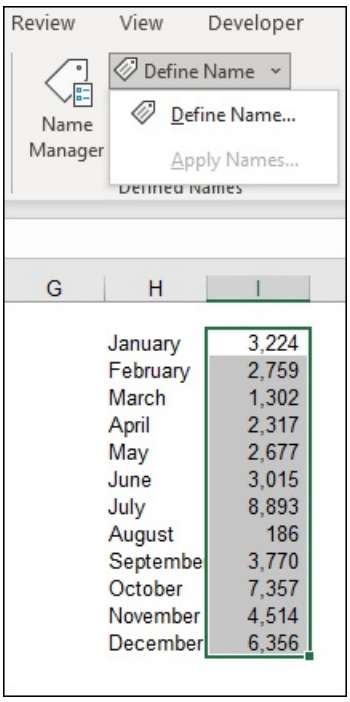


FIGURE 6-1: Click the Define Name command to name a chosen range.

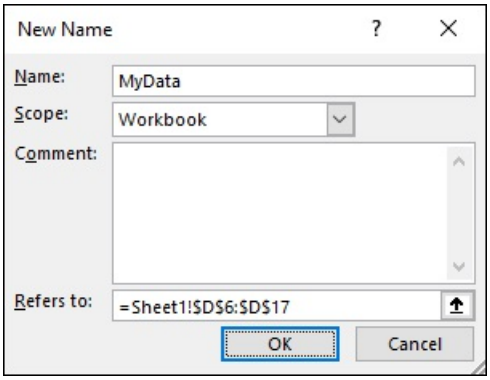


FIGURE 6-2: Give your range a name.

To confirm that your range is now named, click the Name Manager command on the Formulas tab. In the Name Manager dialog box (see Figure 6-3), you can see all the applied named ranges.

Creating a named range via VBA is much less involved. You can directly define the Name property of the Range object:

```
Sub Macro1()

  Range("J6:J17").Name = "MyData"

End Sub
```

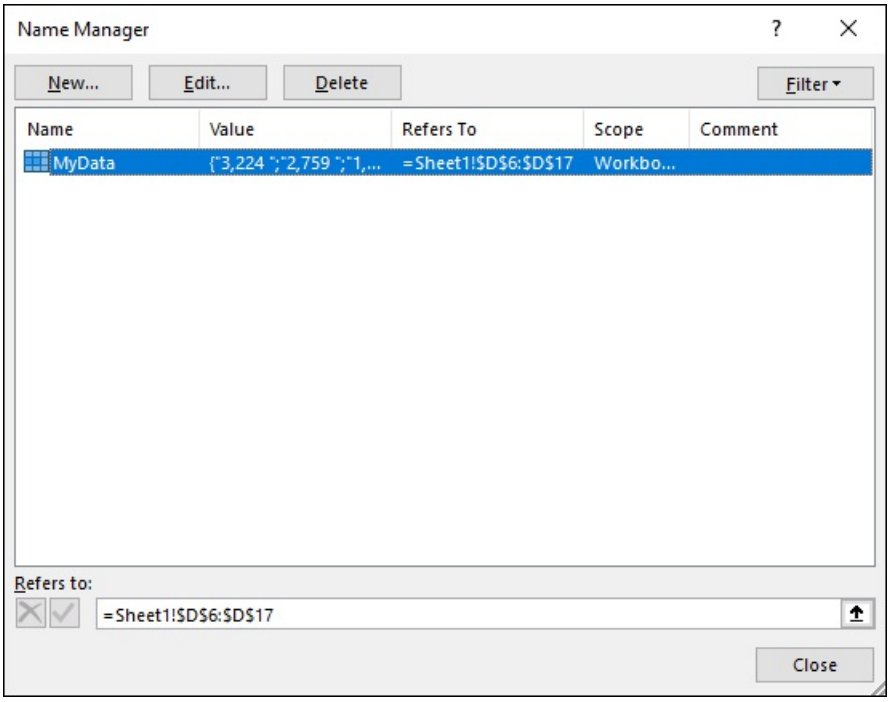


FIGURE 6-3: The Name Manager dialog box lists all the applied named ranges.

Admittedly, you'd be hard pressed to find a situation where you would need to automate the creation of named ranges. The real efficiency comes in manipulating them via VBA.

To reference the range, you simply pass the defined name as an argument to the Range object. This allows you to work with the range, such as selecting it:

```
Sub Macro1()

  Range("MyData").Select

End Sub
```

As with normal ranges, you can refer to the range using the With…End With statement. This statement tells Excel that any action you perform applies to the object to which you've pointed. This not only prevents you from having to repeat syntax, but it also allows for the easy addition of actions by simply adding them between the With and End With statements.

```
Sub Macro1()

  With Range("MyData")
    .NumberFormat = "#,##0"
    .Font.Bold = True
    .Interior.Color = vbYellow
  End With

End Sub
```

To implement this kind of a macro, you can copy and paste it into a standard module, as outlined in the section at the beginning of this chapter, "Installing Macros."

Looping Through a Range of Cells

One must-have VBA skill is the ability to loop through a range. If you do any serious macro

work in Excel, you will soon encounter the need to go through a range of cells one-by-one and perform some action.

This basic macro shows you a simple way to loop through a range.

In this macro, you use two Range object variables. One of the variables holds all the cells you want to loop through and the other, known as the *control* variable, holds each individual cell as you loop through the range. Then you use the For Each statement to assign each cell to the control variable:

```
Sub Macro1()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Define the target range.
  Set MyRange = Range("D6:D17")

'Step 3: Start looping through the range.
  For Each MyCell In MyRange

'Step 4: Do something with each cell.
    If MyCell.Value > 3000 Then
      MyCell.Font.Bold = True
    End If

'Step 5: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as the macro loops through them one by one.
2. Assigns the target range to the MyRange variable. In this example, you're using Range("D6:D17"). If your target range is a named range, you could simply enter its name — Range("MyNamedRange").
3. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
4. If Then keywords evaluate the Value property of the cell currently assigned to MyCell. If it passes the condition, an action is performed — in this case making the cell bold.

   TIP Other actions you might take are deleting rows, inserting rows, or changing the cell's background color.
5. Loops back to the For Each line to assign the next cell in the range to MyCell. After each cell in the target range has been assigned, the macro ends.

To implement this macro, you can copy and paste it into a standard module, as outlined in the first section of this chapter, "Installing Macros."

Inserting Blank Rows in a Range

Occasionally, you may need to dynamically insert rows into your dataset. Although blank rows are generally bothersome, in some situations, the final formatted version of your report requires them to separate data. The macro in this section adds blank rows into a range.

This macro performs a reverse loop through the chosen range using a counter. It starts at the last row of the range by inserting two blank rows, and then moves to the previous row in the range. It keeps doing that same insert for every loop, each time decrementing the counter to the previous row.

```
Sub Macro1()

'Step1: Declare your variables.
  Dim MyRange As Range
  Dim iCounter As Long

'Step 2: Define the target range.
  Set MyRange = Range("C6:D17")

'Step 3: Start reverse looping through the range.
  For iCounter = MyRange.Rows.Count To 2 Step -1

'Step 4: Insert two blank rows.
    MyRange.Rows(iCounter).EntireRow.Insert
    MyRange.Rows(iCounter).EntireRow.Insert

'Step 5: Decrement the counter.
  Next iCounter

End Sub
```

1. Declares two variables, an object variable called MyRange to hold the entire range you want to loop through, and a Long Integer variable called iCounter to serve as control variable.
2. Assigns the target range to the MyRange variable. In this example, you're using Range("C6:D17"). If your target range is a named range, you could simply enter its name — Range("MyNamedRange").
3. Sets the parameters for the For Next loop so it loops from the last cell of the range to the second from the top. The control variable is initially set to the number of rows in the range and decrements down to 2 (the second row of the chosen range). Use the Step keyword to tell VBA how many numbers to skip in each iteration. When a For Next loop goes from high to low, you must specify the Step. When it goes from low to high, the default Step is 1 and you only have to specify it if it's different than the default.
4. Passes the iCounter control variable to the Rows property to refer to a specific row. In the first iteraction, iCounter is 12 and the Rows property returns the twelfth (last) row in the range. The EntireRow method returns the whole row and the Insert method inserts a new blank row. Because you want two blank rows, you do this twice.
5. Loops back to decrement the counter.



TIP You can loop through a range using a For Each or a For Next loop. When you're inserting or deleting cells, the For Each loop can lose its place. For example, if it records that it has 20 cells to loop through and you delete a row, it skips some rows. Always use a For Next loop to loop backward through a range when inserting or deleting cells.

To implement this macro, you can copy and paste it into a standard module, as outlined in the first section of this chapter, "Installing Macros."

Unhiding All Rows and Columns

When you're auditing a spreadsheet that you didn't create, you often want to ensure you're getting a full view of what is exactly in the spreadsheet. To do so, you need to ensure that no columns and rows are hidden. This simple macro automatically unhides all rows and columns for you.

In this macro, you call the Columns collection and the Rows collection of the worksheet. Each collection has properties that dictate where its objects are hidden or visible. Running this macro unhides every column in the Columns collection and every row in the Rows collection.

```
Sub Macro1()

  Columns.EntireColumn.Hidden = False
  Rows.EntireRow.Hidden = False

End Sub
```

The best place to store this macro is in your Personal Macro Workbook. (See the earlier section "Installing Macros.") This way, the macro is always available to you.

Deleting Blank Rows

Work with Excel long enough, and you'll find ut that blank rows can often cause havoc on many levels. They can cause problems with formulas, introduce risk when copying and pasting, and sometimes cause strange behaviors in PivotTables. If you find that you're manually searching forand deleting blank rows in your data sets, this macro can help automate that task.

In this macro, you're using the UsedRange property of the Activesheet object to define the range you're working with. The UsedRange property gives you a range that encompasses the cells that have been used to enter data. You then establish a counter that starts at the last row of the used range to check whether the entire row is empty. If the entire row is indeed empty, you remove the row. You keep doing that same delete for every iteration, each time decrementing the counter to the previous row.

```
Sub Macro1()

'Step1: Declare your variables.
  Dim MyRange As Range
  Dim iCounter As Long
```

```
'Step 2: Define the target range.
  Set MyRange = ActiveSheet.UsedRange

'Step 3: Start reverse looping through the range.
  For iCounter = MyRange.Rows.Count To 1 Step -1

'Step 4: If entire row is empty then delete it.
    If Application.WorksheetFunction.CountA _
      (Rows(iCounter).EntireRow) = 0 Then

      Rows(iCounter).EntireRow.Delete
    End If

'Step 5: Decrement the counter.
  Next iCounter

End Sub
```

1. Declares two variables, an object variable called MyRange to hold the entire range you want to loop through, and a Long Integer variable called iCounter to serve as control variable.
2. Assigns the used range to the MyRange variable by using the UsedRange property. The UsedRange property gives you a range that encompasses the cells that have been used to enter data.



TIP If you want to specify an actual range or a named range, you could simply enter its name — Range(“MyNamedRange”).

3. Sets the parameters for the For Next loop so it loops from the last row of the range to the first. The control variable is initially set to the number of rows in the range and decrements down to 1. The Step keyword tells VBA how many numbers to skip in each iteration. When a For Next loop goes from high to low, you must specify the Step. When it goes from low to high, the default Step is 1 and you only have to specify it if it's different than the default.
4. The COUNTA worksheet function checks whether data is anywhere in the row. You can use Application.WorksheetFunction to access any worksheet function that's available to VBA. Passes the entire row to the function by identifying the current row in the loop (Rows(iCounter)) and using the EntireRow property to expand it for all columns. If the function returns zero, there is no data and you use the Delete method to delete it.
5. Loops back to decrement the counter.

The best place to store this macro is in your Personal Macro Workbook, as outlined in the first section in this chapter, “Installing Macros.” This way, the macro is always available to you.

Deleting Blank Columns

Just as with blank rows, blank columns also have the potential of causing unforeseen errors. If you find that you’re manually searching for and deleting blank columns in your data sets, this macro can automate that task.

In this macro, you're using the UsedRange property of the ActiveSheet object to define the range to work with. The UsedRange property gives you a range that encompasses the cells that have been used to enter data. You then establish a counter that starts at the last column of the used range, checking whether the entire column is empty. If the entire column is indeed empty, you remove the column. You keep doing that same delete for every loop, each time incrementing the counter to the previous column.

```
Sub Macro1()

'Step1: Declare your variables.
  Dim MyRange As Range
  Dim iCounter As Long

'Step 2: Define the target range.
  Set MyRange = ActiveSheet.UsedRange

'Step 3: Start reverse looping through the range.
  For iCounter = MyRange.Columns.Count To 1 Step -1

'Step 4: If entire column is empty then delete it.
    If Application.WorksheetFunction.CountA _
      (Columns(iCounter).EntireColumn) = 0 Then

      Columns(iCounter).EntireColumn.Delete
    End If

'Step 5: Decrement the counter.
    Next iCounter

End Sub
```

1. Declares two variables, an object variable called MyRange to hold the entire range you want to loop through, and a Long Integer variable called iCounter to serve as control variable.
2. Assigns the used range to the MyRange variable with the UsedRange property. The UsedRange property gives you a range that encompasses the cells that have been used to enter data. Note that if you want to specify an actual range or a named range, you can simply enter its name — Range("MyNamedRange").
3. Sets the parameters for the For Next loop so it loops from the last column of the range to the first. The control variable is initially set to the number of columns in the range and decrements down to 1. The Step keyword tell VBA how many numbers to skip in each iteration. When a For Next loop goes from high to low, you must specify the Step. When it goes from low to high, the default Step is 1 and you only have to specify it if it's different than the default.
4. The COUNTA worksheet function checks whether data is anywhere in the column. You can use Application.WorksheetFunction to access any worksheet function that's available to VBA. Passes the entire column to the function by identifying the current column in the loop (Columns(iCounter)) and using the EntireColumn property to expand it for all rows. If the function returns zero, there is no data and you use the Delete method to delete it.
5. Loops back to decrement the counter.

The best place to store this macro is in your Personal Macro Workbook, as outlined in the "Installing Macros" section, at the beginning of this chapter. This way, the macro is always

available to you.

Limiting Range Movement to a Particular Area

Excel gives you the ability to limit the range of cells that a user can scroll through. The macro shown in this section is something you can easily implement.

Excel's ScrollArea property allows you to set the scroll area for a particular worksheet. For example, this statement sets the scroll area on Sheet1 so the user can't activate any cells outside of A1:M17.

```
Sheets("Sheet1").ScrollArea = "A1:M17"
```

Because this setting is not saved with a workbook, you have to reset it each time the workbook is opened. You can accomplish this by implementing this statement in the Workbook_Open event:

```
Private Sub Workbook_Open()

  Me.Sheets("Sheet1").ScrollArea = "A1:M17"

End Sub
```

The Me keyword refers to the object whose module the code is in. In this case, it's the ThisWorkbook module, so Me refers to ThisWorkbook. If for some reason you need to clear the scroll area limits, you can remove the restriction with this statement:

```
Me.Sheets("Sheet1").ScrollArea = ""
```

To implement this macro, you need to copy and paste it into the Workbook_Open event Code pane, as shown in Figure 6-4. (See the earlier section "Installing Macros." Placing the macro here allows it to run each time the workbook opens.

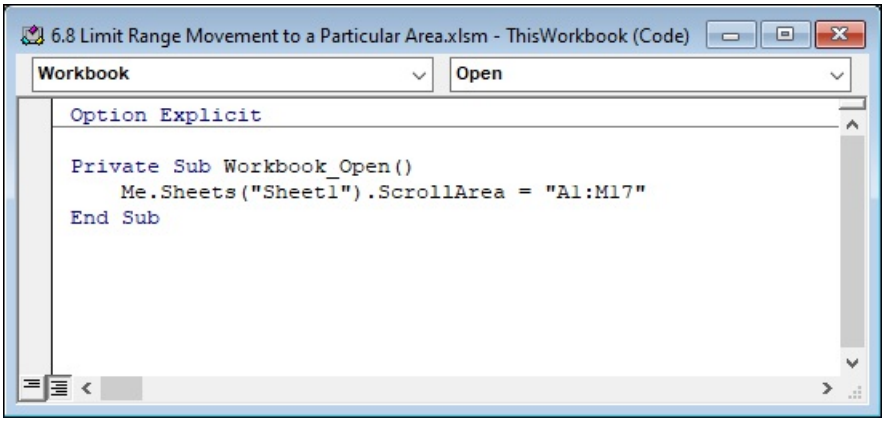


FIGURE 6-4: Type or paste your code in the Workbook Open event Code pane.

Selecting and Formatting All Formulas in a Workbook

When auditing an Excel workbook, it's paramount to have a firm grasp of all the formulas in each sheet. This means finding all the formulas, which can be an arduous task if done manually.

However, Excel provides a slick way of finding and tagging all the formulas on a worksheet. The macro in this section exploits this functionality to dynamically find all cells that contain formulas.

Excel has a set of predefined "special cells" that you can select by using the Go To Special dialog box. To select special cells manually, go to the Home tab on the Ribbon, select Go To Special from the Find & Select drop-down list (or press F5). Figure 6-5 shows the Go To Special dialog box.

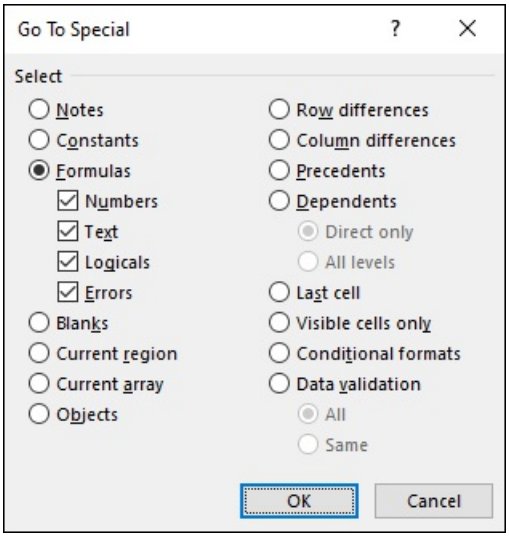


FIGURE 6-5: The Go To Special dialog box.

Here, you can select a set of cells based on a few defining attributes. One of those defining attributes is formulas. Selecting the Formulas option effectively selects all cells that contain formulas (see Figure 6-6). At this point, you can color the cells to indicate they contain a formula.



	C	D	E	F	G	H	I	J	K	L
5										
6	$719.20	6	$119.87							
7										
8					$867.59	6	$144.60			
9										
10										
11										
12	$479.46	4	$ 79.91							
13										
14										
15					$578.40	4	$ 96.40			
16										
17										
18										
19										
20								$2,210.24	12	$368.37
21										
22										
23			$722.99	5	$120.50					
24										

FIGURE 6-6: Choosing Formulas in the Go To Special dialog box tells Excel to select all cells containing a formula.

This macro programmatically does the same thing for the entire workbook at the same time. Here, you're using the SpecialCells method of the Cells collection. The SpecialCells method requires a Type parameter that represents the type of special cell. In this case, you use xlCellTypeFormulas.

In short, you're referring to a special range that consists only of cells that contain formulas. You refer to this special range using the With…End With statement. This statement tells Excel that any action you perform applies only to the range to which you've pointed. Here, you're coloring the interior of the cells in the chosen range.

```
Sub Macro1()

'Step 1: Declare your variables.
  Dim ws As Worksheet

'Step 2: Avoid error if no formulas are found.
  On Error Resume Next

'Step 3: Start looping through worksheets
  For Each ws In ActiveWorkbook.Worksheets

'Step 4: Select cells and highlight them.
    With ws.Cells.SpecialCells(xlCellTypeFormulas)
      .Interior.Color = vbYellow
    End With

'Step 5: Get next worksheet.
  Next ws

End Sub
```

1. Declares an object variable called ws. This creates a memory container for each worksheet the macro loops through.
2. If no formulas are in the spreadsheet, VBA throws an error. To avoid the error, tells VBA to continue with the macro if an error is triggered.
3. Begins looping, telling VBA to evaluate all worksheets in the active workbook.

4. Sets the Interior.Color propery all cells containing formulas to the built-in constant vbYellow.
5. Loops back to get the next sheet. After all of the sheets are evaluated, the macro ends.

The best place to store this macro is in your Personal Macro Workbook, as outlined in the "Installing Macros" section, at the beginning of this chapter. This way, the macro is always available.

Finding and Selecting the First Blank Row or Column

You may run across scenarios where you have to append rows or columns to an existing data set. When you need to append rows, you need to be able to find the last used row and then move down to the next empty cell (shown in Figure 6-7). Likewise, in situations where you need to append columns, you need to be able to find the last used column and then move over to the next empty cell.

	A	B	C	D	E	F	G	H
5		January	February	March	April	May	June	July
6	Product 1	72,542	70,916	49,289	3,538	87,442	61,690	51,382
7	Product 2	28,187	18,175	71,645	99,211	10,516	91,078	71,116
8	Product 3	75,043	8,280	24,234	40,255	77,472	85,502	82,010
9	Product 4	4,984	31,805	47,905	45,292	89,648	94,801	10,890
10	Product 5	42,680	47,574	35,982	18,860	56,353	91,804	69,317
11	Product 6	16,140	3,676	76,712	27,619	68,199	36,281	10,567
12	Product 7	97,001	56,895	40,052	79,893	78,703	40,253	30,288
13	Product 8	21,227	28,168	97,923	16,585	1,843	98,599	34,916
14	Product 9	56,692	17,489	82,649	28,960	68,233	21,507	33,244
15	Product 10	64,906	54,698	93,271	29,388	29,712	54,326	27,516
16	Product 11	68,672	29,475	58,379	16,282	2,953	69,435	3,788
17	Product 12	38,676	1,457	3,833	98,225	99,695	2,712	74,492
18								
19								

FIGURE 6-7: You can use a macro to tell Excel to dynamically find the first available cell in a row or column.

The macros in this section allow you to dynamically find and select the first blank row or column. They both use the Cells property and the Offset property as key navigation tools.

The Cells property provides a handy way of selecting ranges through code. It requires only relative row and column positions as parameters. Cells (5,4) translates to row 5, column 4 (or Cell D5). Cells (16, 4) translates to row 16, column 4 (or cell D16). In addition to passing hard numbers to the Cells item, you can also pass expressions.

Cells(Rows.Count, 1) is the same as selecting the last row in the spreadsheet and the first column in the spreadsheet, or cell A1048576.

Cells(1, Columns.Count) is the same as selecting the first row in the spreadsheet and the last column in the spreadsheet, or cell XFD1.

Combining the Cells statement with the End property allows you to jump to the last used row or column. This statement is equivalent to going to cell A1048576 and pressing Ctrl+↑. When you run this statement, Excel automatically jumps to the last used row in column A:

```
Cells(Rows.Count, 1).End(xlUp).Select
```

Running this statement is equivalent to going to cell XFD1 and pressing Ctrl+←. This gets you to the last used column in row 1:

```
Cells(1, Columns.Count).End(xlToLeft).Select
```

When you get to the last used row or column, you can use the Offset property to move down or over to the next blank row or column. The Offset property uses a row and column index to refer to a range a specified number of rows or columns away from the starting range. For example, this statement selects cell A2 because the row index in the offset is referring to the range one row down:

```
Range("A1").Offset(1, 0).Select
```

This statement selects cell C4 because the row and column indexes refer to a range three rows down and two columns right:

```
Range("A1").Offset(3, 2).Select
```

Pulling all these concepts together, you can create a macro that selects the first blank row or column. This macro selects the first blank row:

```
Sub Macro1()

'Step 1: Declare your variables.
  Dim LastRow As Long

'Step 2: Capture the last used row number.
  LastRow = Cells(Rows.Count, 1).End(xlUp).Row

'Step 3: Select the next row down.
  Cells(LastRow, 1).Offset(1, 0).Select

End Sub
```

1. Declares a Long Integer variable called LastRow to hold the row number of the last used row.
2. Captures the last used row by starting at the very last row in the worksheet and using the End property to jump to the first non-empty cell (the equivalent of going to cell A1048576 and pressing Ctrl+↑).
3. Uses the Offset property to move down one row and selects the first blank cell in column A.

This macro selects the first blank column:

```
Sub Macro1()

'Step 1: Declare your variables.
  Dim LastColumn As Long

'Step 2: Capture the last used column number.
  LastColumn = Cells(5, Columns.Count).End(xlToLeft).Column

'Step 3: Select the next column over.
  Cells(5, LastColumn).Offset(0, 1).Select
```

```
End Sub
```

1. Declares a Long Integer variable called LastColumn to hold the column number of the last used column.
2. Captures the last used column by starting at the very last column in the worksheet and using the End property to jump to the first non-empty column (the equivalent of going to cell XFD5 and pressing Ctrl+←).
3. The Offset property moves over one column and selects the first blank column in row 5.

You can implement these macros by pasting them into a standard module, as outlined in the first section of this chapter, "Installing Macros."

Chapter 7

Manipulating Data with Macros

IN THIS CHAPTER

- **Copying and pasting a range and converting formulas in a range**
- **Performing text to columns on all columns**
- **Converting trailing minus signs and padding cells with zeros**
- **Truncating postal codes to the left five**
- **Appending text to the left or right of your cells**
- **Cleaning up data including duplicates, extra cell space, and blank cells**
- **Selectively hiding AutoFilter dropdowns**
- **Copying filtered rows and showing filtered columns in status bar**

When working with information in Excel, you often have to transform the data in some way. Transforming it generally means cleaning, standardizing, or shaping data in ways that are appropriate for your work. This can mean anything from cleaning out extra spaces, to padding numbers with zeros, to filtering data for certain criteria.

This chapter shows you some of the more useful macros you can use to dynamically transform the data in your workbooks. If you want, you can combine these macros into one, running each piece of code in a sequence that essentially automates the scrubbing and shaping of your data.

Installing Macros

You can install the macros in this chapter in three places depending on what kind of macro it is and how available you want the macro to be. Event macros are put into the module of the object the event is tied to. Non-event macros go into a standard module and can be in the workbook in which they are meant to run or in a workbook called the Personal Macro Workbook if you want to be able to run them on any workbook.

Event macros

In VBA, objects have events. VBA “listens” for these events to happen and runs special event macros when it detects and event. For example, the workbook object has an Open event. When VBA detects that the workbook has been opened, it looks for a procedure called Workbook_Open in the ThisWorkbook module. If it's there, it runs that macro.

Most of the event macros that you'll use, and most of the examples in the book, are workbook or worksheet events. Workbook event macros go in the ThisWorkbook module and worksheet event macros go in the module for the sheet you want the event code to run on.

To install an event macro, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **In the Project Explorer, find your project/workbook name and click the plus sign next to it to see all the objects.**
3. **Double-click ThisWorkbook.**
4. **Select the event in the right drop-down list.**
5. **Type or paste the code in the module.**

Personal Macro Workbook

There is a special workbook called the Personal Macro Workbook named personal.xlsb. If you record a macro, you can choose this workbook as the destination of the recorded macro. The workbook doesn't exist until you first record a macro into it. After that, it lives in a folder and Excel loads that workbook every time it starts.

The benefit of a workbook that opens every time Excel starts is that all the macros in that workbook are always available to you. If you have a macro that works on the active sheet or whatever range is currently selected, that's a good candidate for the Personal Macro Workbook.

To install a macro in the Personal Macro Workbook, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click personal.xlb in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**



REMEMBER If you don't see personal.xlsb in the Project Explorer, record a macro to the Personal Macro Workbook and Excel will create it for you.

Standard macros

All the other macro examples in this book are installed in a specific workbook in a standard module. If it's not an event macro and it's not a macro you want in the Personal Macro Workbook, then install it using the following steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click the project/workbook name in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**

Copying and Pasting a Range

One of the basic data manipulation skills you'll need is copying and pasting a range of data. It's fairly easy to do this manually. Luckily, it's just as easy to copy and paste via VBA.

In this macro, you use the Copy method of the Range object to copy data from D6:D17 and paste it into L6:L17. Note the use of the Destination argument. This argument tells Excel where to paste the data.

```
Sub CopyAndPasteRange()

  Sheets("Sheet1").Range("D6:D17").Copy _
    Destination:=Sheets("Sheet1").Range("L6:L17")

End Sub
```

When working with your spreadsheet, you likely often have to copy formulas and paste them as values. To do this in a macro, you can use the PasteSpecial method. In this example, you copy the formulas in F6:F17 to M6:M17. Notice that you're not only pasting as values using xlPasteValues, but you are also using xlPasteFormats to apply the formatting from the copied range.

```
Sub CopyAndPasteRange()

  Sheets("Sheet1").Range("F6:F17").Copy
  Sheets("Sheet1").Range("M6:M17").PasteSpecial xlPasteValues
  Sheets("Sheet1").Range("M6:M17").PasteSpecial xlPasteFormats

End Sub
```



REMEMBER The ranges specified for this macro are for demonstration purposes. Alter the ranges to suit the data in your worksheet.

To implement this macro, you can copy and paste it into a standard module, as outlined in the previous section.

Converting All Formulas in a Range to Values

In some situations, you may want to apply formulas in a certain workbook, but you don't necessarily want to keep or distribute the formulas with your workbook. In these situations, you may want to convert all the formulas in a given range to values.

In this macro, you use two Range object variables. One of the variables captures the scope of data you're working with and the other is used to hold each individual cell as you go through the range. Then you use the For Each statement to work with each cell in the target range. In every iteration of the loop, you check to see whether the cell contains a formula. If it does, you replace the formula with the value shown in the cell.

```
Sub CovertFormulasToValues()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range
```

```
'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't undo this action. " & _
        "Save workbook first?", vbYesNoCancel)
    Case vbYes
      ThisWorkbook.Save

    Case vbCancel
      Exit Sub
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: If cell has formula, set to the value shown.
    If MyCell.HasFormula Then
      MyCell.Value = MyCell.Value
    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Shows a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   TIP When you run a macro, it erases the undo stack. This means you can't undo the changes a macro makes. Because you're changing data, give yourself the option of saving the workbook before running the macro.
3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").
4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
5. Checks whether the cell in this iteration contains a formula using the HasFormula property. If it does, the cell's Value property is set equal to itself. This replaces the formula with a hard-coded value.
6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.

To implement this macro, you can copy and paste it into a standard module, as described in the first section of this chapter, "Installing Macros."

Text to Columns on All Columns

When you import data from other sources, you may sometimes wind up with cells where the number values are formatted as text. You typically recognize this problem because no matter what you do, you can't format the numbers in these cells to numeric, currency, or percentage formats. You may also see a smart tag on the cells (see Figure 7-1) that tells you the cell is formatted as text.

It's easy enough to fix this manually by clicking the Text to Columns command on the Data tab (Figure 7-2). This opens the Text to Columns dialog box shown in Figure 7-3. There is no need to go through all the steps in this wizard; simply click the Finish button to apply the fix.

Again, this is a fairly simple action. The problem, however, is that Excel doesn't let you perform the Text to Columns fix on multiple columns. You have to apply this fix one column at a time. This can be a real nuisance if you have this issue in many columns.

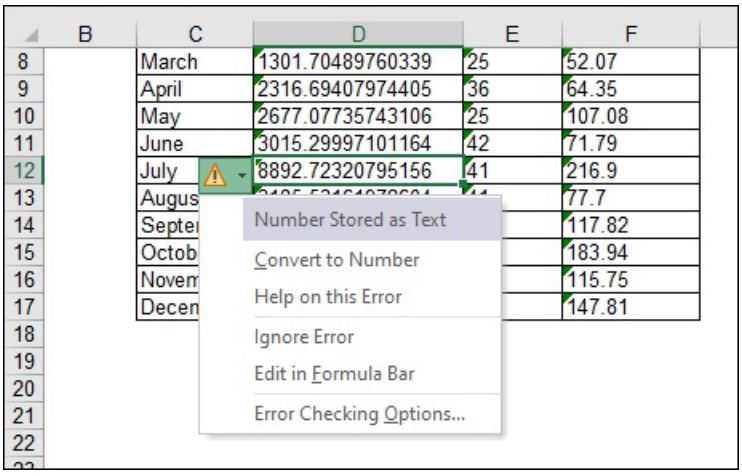


FIGURE 7-1: Imported numbers are sometimes formatted as text.

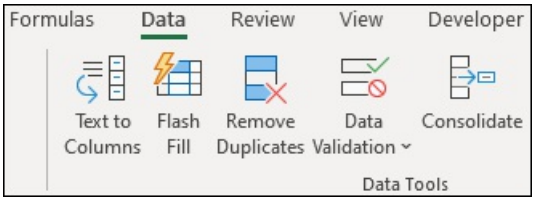


FIGURE 7-2: Click the Text to Columns command.

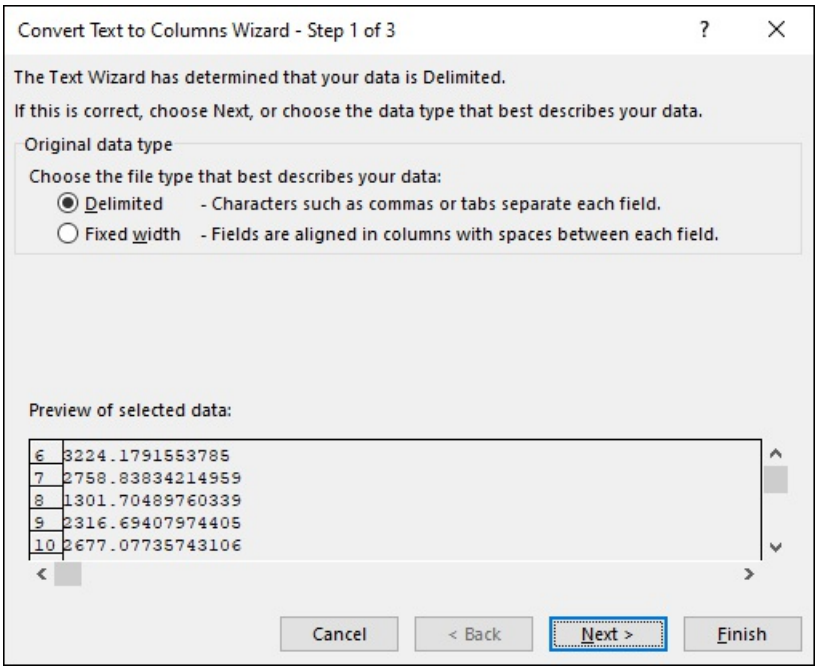


FIGURE 7-3: Clicking Finish in the Text to Columns dialog box corrects incorrectly formatted numbers.

Here is where a simple macro can help you save your sanity. In this macro, you use two Range object variables to go through your target range, using the For Each statement to assign each cell in the target range. Every time a cell is assigned, you reset the value of the cell. This has the same effect as the Text to Columns command.

```
Sub TTCAllColumns()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't Undo this action. " & _
            "Save Workbook First?", vbYesNoCancel)
    Case Is = vbYes
      ThisWorkbook.Save

    Case Is = vbCancel
      Exit Sub
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: Reset the cell value.
    If Not IsEmpty(MyCell) Then
      MyCell.Value = MyCell.Value
    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one

by one.

2. Shows a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   Remember that when you run a macro, it erases the undo stack. This means you can't undo the changes a macro makes. Because you are changing data, give yourself the option of saving the workbook before running the macro.

3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   REMEMBER You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").

4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.

5. After a cell is assigned, the IsEmpty function makes sure the cell is not empty. This saves a little time by skipping the cell if there is nothing in it. Then simply set the cell to its own value to force Excel to reevaluate whether the data is numeric.

6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.

To implement this macro, you can copy and paste it into a standard module. See the earlier section, "Installing Macros," for the details.

Converting Trailing Minus Signs

Legacy and mainframe systems are notorious for outputting trailing minus signs. In other words, instead of a number such as -142, some systems output 142-. This obviously wreaks havoc on your spreadsheet — especially if you need to perform mathematic operations on the data. This macro goes through a target range and fixes all the negative minus signs so that they show up in front of the number rather than the end.

In this macro, you use two Range object variables to go through your target range, using the For Each statement to assign each cell in the target range. Every time a cell is assigned, you convert the value of the cell into a Double numeric data type by using the Cdbl function. The Double data type forces any negative signs to show at the front of the number.

```
Sub FixTrailingMinus()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't Undo this action. " & _
            "Save Workbook First?", vbYesNoCancel)
```

```
    Case Is = vbYes
      ThisWorkbook.Save

    Case Is = vbCancel
      Exit Sub
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: Convert the value to a double.
    If IsNumeric(MyCell) Then
      MyCell.Value = CDbl(MyCell.Value)
    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Show a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   Remember that when you run a macro, it erases the undo stack. This means you can't undo the changes a macro makes. Because you are changing data, give yourself the option of saving the workbook before running the macro.
3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").
4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
5. After a cell is assigned, the IsNumeric function checks whether the value can be evaluated as a number. This is to ensure you don't affect text fields. Then the cell's value is passed through the Cdbl function. This converts the value to the Double numeric data type, forcing the minus sign to the front.
6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.



TIP Because you define the target range as the current selection, you want to be sure to select the area where your data exists before running this code. In other words, you wouldn't want to select the entire worksheet. Otherwise, every empty cell in the spreadsheet would be filled with a zero. Of course, you can ensure this is never a problem

by explicitly defining the target range, such as Set MyRange = Range(“A1:Z100”).

To implement this macro, you can copy and paste it into a standard module. The first section in this chapter, “Installing Macros,” details how to do run this macro.

Trimming Spaces from All Cells in a Range

A frequent problem when you import data from other sources is leading or trailing spaces. That is, the imported values have spaces at the beginning or end of the cell. This obviously makes it difficult to do things such as VLOOKUP or sorting. Here is a macro that makes it easy to search for and remove extra spaces in your cells.

In this macro, you enumerate through a target range, passing each cell in that range through the Trim function.

```
Sub TrimSpaces()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't Undo this action. " & _
            "Save Workbook First?", vbYesNoCancel)
    Case Is = vbYes
      ThisWorkbook.Save

    Case Is = vbCancel
      Exit Sub
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: Trim the spaces.
    If Not IsEmpty(MyCell) Then
      MyCell = Trim(MyCell)
    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Shows a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   Remember that when you run a macro, it erases the undo stack. This means you can’t undo the changes a macro makes. Because you are changing data, give yourself the option of saving the workbook before running the macro.
3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

You can easily set the MyRange variable to a specific range such as Range(“A1:Z100”). Also, if your target range is a named range, you could simply enter its name: Range(“MyNamedRange”).

4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
5. After a cell is assigned, the IsEmpty function makes sure the cell is not empty. This saves a little time by skipping the cell if there is nothing in it. The value of that cell passs to the Trim function. The Trim function is a built-in VBA function that removes leading and trailing spaces.
6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.



TIP Instead of VBA's Trim, you can use Application.WorksheetFunction.Trim to use Excel's TRIM function. Excel's function removes leading and trailing spaces as well as any duplicate spaces within the text.

To implement this macro, you can copy and paste it into a standard module. See the section at the beginning of this chapter, “Installing Macros,” for how to do that.

Truncating Zip Codes to the Left Five

U.S. zip codes are either five or nine digits. Some systems output a nine-digit zip code, which, for the purposes of a lot of Excel analysis, is too many. A common data standardization task is to truncate zip codes to the left five digits. You might use a formula to do this, but if you are constantly cleaning up your zip codes, the macro outlined in this section can help automate that task.



REMEMBER Although this macro solves a specific problem, the concept of truncating data remains useful for many other types of data cleanup activities.

This macro uses the Left function to extract the left five characters of each zip code in the given range.

```
TruncateZip

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't Undo this action. " & _
            "Save Workbook First?", vbYesNoCancel)
    Case Is = vbYes
      ThisWorkbook.Save

    Case Is = vbCancel
      Exit Sub
```

```
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: Extract out the left 5 characters.
    If Not IsEmpty(MyCell) Then
      MyCell = Left(MyCell, 5)
    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Shows a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   Remember that when you run a macro, it erases the undo stack. This means you can't undo the changes a macro makes. Because you're changing data, give yourself the option of saving the workbook before running the macro.
3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").
4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
5. After a cell is assigned, the IsEmpty function makes sure the cell is not empty. This saves a little time by skipping the cell if there is nothing in it. The cell's value passes through the Left function. The Left function allows you to extract out the *n*th leftmost characters in a string. In this case, you need left five characters to truncate the zip code to five digits.
6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.



TIP As you may have guessed, you can also use the Right function to extract the *n*th rightmost characters in a string. As an example, it's not uncommon to work with product numbers where the first few characters hold a particular attribute or meaning, whereas the last few characters point to the actual product (as in 100-4567). You can extract the actual product by using Right(Product_Number, 4).



REMEMBER Because the target range is defined as the current selection, you need to select the area where your data exists before running this code. In other words, you wouldn't want to select cells that don't conform to the logic you placed in this macro. Otherwise, every cell you select is truncated, whether you mean it to be or not. Of course, you can ensure this is never a problem by explicitly defining the target range, such as Set MyRange = Range("A1:Z100").

To implement this macro, you can copy and paste it into a standard module, as outlined in the first section of this chapter, "Installing Macros."

Padding Cells with Zeros

Many systems require unique identifiers (such as customer number, order number, or product number) to have a fixed character length. For example, you frequently see customer numbers that look like this: 00000045478. This concept of taking a unique identifier and forcing it to have a fixed length is typically referred to as *padding*. The number is padded with zeros to achieve the prerequisite character length.

It's a pain to do this manually in Excel. However, with a macro, padding numbers with zeros is a breeze.



TIP Some Excel gurus are quick to point out that you can apply a custom number format to pad numbers with zeros by going to the Format Cells dialog box, selecting Custom on the Number tab, and entering "0000000000" as the custom format.

The problem with this solution is that the padding you get is cosmetic only. A quick glance at the Formula bar reveals that the data actually remains numeric without the padding (it does not become textual). So if you copy and paste the data into another platform or non-Excel table, you lose the cosmetic padding.

If all your customer numbers need to be ten characters long, you need to pad the number with enough zeros to get it to ten characters. This macro does just that.

As you review this macro, keep in mind that you need to change the padding logic in Step 5 to match your situation.

```
Sub PadWithZeros()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't Undo this action. " & _
            "Save Workbook First?", vbYesNoCancel)
    Case Is = vbYes
      ThisWorkbook.Save
```

```
    Case Is = vbCancel
      Exit Sub
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: Pad with ten zeros then take the right 10.
    If Not IsEmpty(MyCell) Then

      MyCell.NumberFormat = "@"
      MyCell = String(10, "0") & MyCell
      MyCell = Right(MyCell, 10)

    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Shows a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   Remember that when you run a macro, it erases the undo stack. This means you can't undo the changes a macro makes. Because you are changing data, give yourself the option of saving the workbook before running the macro.
3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").
4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
5. After a cell is assigned, the IsEmpty function makes sure the cell isn't empty. This saves a little time by skipping the cell if there is nothing in it. Ensures the cell is formatted as text so Excel doesn't just drop the leading zeros as soon as you add them by setting the NumberFormat property to the @ symbol. This symbol indicates text formatting.

   Concatenates the cell value with ten zeros using the String function, which repeats a character a specified number of times. The ampersand (&) operator concatentates two strings.

   The Right function returns the ten rightmost characters. This effectively gives the cell value, padded with enough zeros to make ten characters.
6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.

To implement this macro, you can copy and paste it into a standard module. See the first section of this chapter, "Installing Macros," for more information.

Replacing Blanks Cells with a Value

In some analyses, blank cells can cause all kinds of trouble. They can cause sorting issues, they can prevent proper auto filling, they can cause your PivotTables to apply the Count function rather than the Sum function, and so on.

Blanks aren't always bad, but if they are causing you trouble, you can use this macro to quickly replace the blanks in a given range with a value that indicates a blank cell.

This macro enumerates through the cells in the given range and then uses the Len function to check the length of the value in the assigned cell after it's been trimmed of blank spaces. Trimmed, blank cells have a character length of 0. If the length is indeed 0, the macro enters a 0 in the cell, replacing the blanks. A 0 may not be the appropriate indicator of a blank cell for your specific circumstance, so choose another value if necessary.

```
Sub ReplaceBlankCells()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't Undo this action. " & _
            "Save Workbook First?", vbYesNoCancel)
    Case Is = vbYes
      ThisWorkbook.Save

    Case Is = vbCancel
      Exit Sub
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: Ensure the cell has text formatting.
    If Len(Trim(MyCell.Value)) = 0 Then
      MyCell.Value = 0
    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Shows a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   Remember that when you run a macro, it erases the undo stack. This means you can't undo the changes a macro makes. Because you are changing data, give yourself the option

of saving the workbook before running the macro.

3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").

4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.

5. After a cell is assigned, the Len and Trim functions check whether the value should be replaced.

   The Len function is a VBA function that returns a number corresponding to the length of the string being evaluated. If the cell is blank or only contains spaces, the length will be 0; at which point, the macro replaces the cell value with a 0. You could obviously replace the blank with any value you want (N/A, TBD, No Data, and so on).

6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.

   REMEMBER Because the target range is defined as the current selection, you want to be sure to select the area where your data exists before running this code. That is to say, you wouldn't want to select the entire worksheet. Otherwise, every empty cell in the spreadsheet would be filled with a zero. Of course, you can ensure this is never a problem by explicitly defining a range, such as Set MyRange = Range("A1:Z100").

To implement this macro, you can copy and paste it into a standard module, as outlined in the first section of this chapter, "Installing Macros."

Appending Text to the Left or Right of Your Cells

Every so often, you come upon a situation where you need to attach data to the beginning or end of the cells in a range. For example, you may need to add an area code to a set of phone numbers. The macro demonstrates how you can automate the data standardization tasks that require appending data to values.

This macro uses two Range object variables to go through the target range, leveraging the For Each statement to activate each cell in the target range. Every time a cell is activated, the macro attaches an area code to the beginning of the cell value.

```
Sub AddText()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range
```

```
'Step 2: Save the workbook before changing cells?
  Select Case MsgBox("Can't Undo this action. " & _
            "Save Workbook First?", vbYesNoCancel)
    Case Is = vbYes
      ThisWorkbook.Save

    Case Is = vbCancel
      Exit Sub
  End Select

'Step 3: Define the target range.
  Set MyRange = Selection

'Step 4: Start looping through the range.
  For Each MyCell In MyRange

'Step 5: Ensure the cell has text formatting.
    If Not IsEmpty(MyCell.Value) Then
      MyCell.Value = "(972) " & MyCell.Value
    End If

'Step 6: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Shows a message box that gives you three choices: Yes, No, and Cancel. Clicking Yes saves the workbook and continues with the macro. Clicking Cancel exits the procedure without running the macro. Clicking No runs the macro without saving the workbook.

   Remember that when you run a macro, it erases the undo stack. This means you can't undo the changes a macro makes. Because you are changing data, give yourself the option of saving the workbook before running the macro
3. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").
4. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
5. After a cell is assigned, ampersand (&) combine an area code with the cell value. If you need to append text to the end of the cell value, you would simply place the ampersand and the text at the end. For example, MyCell.Value = MyCell.Value & "Added Text".
6. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.

To implement this macro, you can copy and paste it into a standard module. See the earlier section, "Installing Macros," in this chapter for more details.

Cleaning Up Non-Printing Characters

Sometimes your data includes non-printing characters, such as line feeds, carriage returns, and non-breaking spaces. These characters often need to be removed before you can use the data for serious analysis.

Now, anyone who has worked with Excel for more than a month knows about the Find and Replace function. You may have even recorded a macro while performing a Find and Replace (a recorded macro is an excellent way to automate your find and replace procedures). So your initial reaction may be to simply find and replace these characters. The problem is that these non-printing characters are for the most part invisible and thus difficult to clean up with the normal Find and Replace routines. The easiest way to clean them up is through VBA.

If you find yourself struggling with those pesky invisible characters, use this general-purpose macro to find and remove all the non-printing characters.

This macro is a relatively simple Find and Replace routine. You are using the Replace method, telling Excel what to find and what to replace it with. This is similar to the syntax you would see when recording a macro while manually performing a Find and Replace.

The difference is that instead of hard-coding the text to find, this macro uses character codes to specify your search text.

Every character has an underlying ASCII code, similar to a serial number. For example, the lowercase letter *a* has an ASCII code of 97. The lowercase letter *c* has an ASCII code of 99. Likewise, invisible characters also have a code:

- The line feed character code is 10.
- The carriage return character code is 13.
- The non-breaking space character code is 160.

This macro uses the Replace method, passing each character's ASCII code as the search item. Each character code is then replaced with an empty string:

```
Sub CleanCells()

'Step 1: Remove line feeds.
  ActiveSheet.UsedRange.Replace What:=Chr(10), Replacement:=""

'Step 2: Remove carriage returns.
  ActiveSheet.UsedRange.Replace What:=Chr(13), Replacement:=""

'Step 3: Remove non-breaking spaces.
  ActiveSheet.UsedRange.Replace What:=Chr(160), Replacement:=""

End Sub
```

1. Looks for and removes the line feed character. The code for this character is 10. You can identify the code 10 character by passing it through the Chr function. After Chr(10) is identified as the search item, an empty string is passed to the Replacement argument.

   Note the use of ActiveSheet.UsedRange. This tells Excel to look in all the cells that have had data entered into them. You can replace the UsedRange object with an actual range if needed.
2. Finds and removes the carriage return character.

3. Finds and removes the non-breaking spaces character.



TIP The characters covered in this macro are only a few of many non-printing characters. However, these are the ones you most commonly run into. If you work with others, you can simply add a new line of code, specifying the appropriate character code. You can enter "ASCII Code Listing" in any search engine to see a list of the codes for various characters.

To implement this macro, you can copy and paste it into a standard module, as outlined in the first section of this chapter, "Installing Macros."

Highlighting Duplicates in a Range of Data

Ever wanted to expose the duplicate values in a range? The macro in this section does just that. There are many manual ways to find and highlight duplicates — ways involving formulas, conditional formatting, sorting, and so on. However, all these manual methods take setup and some level of maintenance as the data changes.

This macro simplifies the task, allowing you to find and highlight duplicates in your data with a click of the mouse (see Figure 7-4).

Customers	Product Number
0000011112	C5567
0000046047	P8844
0000046047	R7609
0000047329	P8895
0000056510	P8867
0000058682	M2244
0000058682	C3322
0000058682	R7786
0000086362	M7765
0000086362	C8874
0000089129	M3345
0000090210	C5521

FIGURE 7-4: This macro dynamically finds and highlights the duplicate values in a selected range.

This macro enumerates through the cells in the target range, leveraging the For Each statement to activate each cell one at a time. It then uses the CountIf function to count the number of times the value in the active cell occurs in the range selected. If that number is greater than 1, it formats the cell yellow.

```
Sub HighlightDupes()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Define the target range.
  Set MyRange = Selection

'Step 3: Start looping through the range.
  For Each MyCell In MyRange
```

```
'Step 4: Color the cell if there's a duplicate.
    If WorksheetFunction.CountIf(MyRange, MyCell.Value) > 1 Then
      MyCell.Interior.Color = vbYellow
    End If

'Step 5: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one-by-one.
2. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range("A1:Z100"). Also, if your target range is a named range, you could simply enter its name: Range("MyNamedRange").
3. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.
4. The WorksheetFunction object provides a way to run many of Excel's spreadsheet functions in VBA. The WorksheetFunction object runs a CountIf function in VBA.

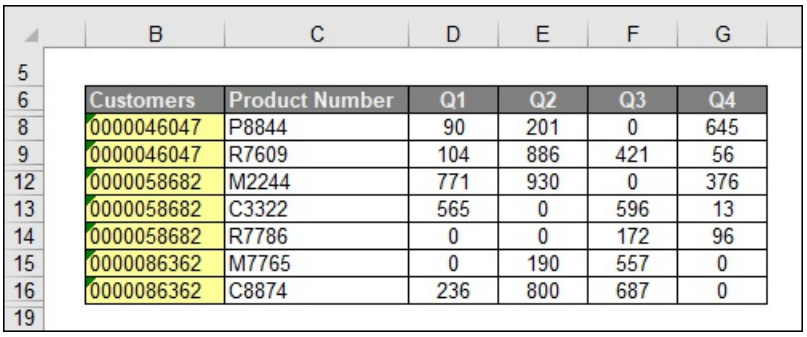
   In this case, you're counting the number of times the assigned cell value (MyCell.Value) is found in the given range (MyRange). If the CountIf expression evaluates to greater than 1, the macro changes the interior color of the cell.
5. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.

To implement this macro, you can copy and paste it into a standard module. The first section in this chapter, "Installing Macros" has more information.

Hiding All but Rows Containing Duplicate Data

With the previous macro, you can quickly find and highlight duplicates in your data. This in itself can be quite useful. But if you have many records in your range, you may want to take the extra step of hiding all the non-duplicate rows.

Take the example in Figure 7-5. Note that only the rows that contain duplicate values are visible. This more readily exposes the duplicate values because they are the only rows showing.



Customers	Product Number	Q1	Q2	Q3	Q4
0000046047	P8844	90	201	0	645
0000046047	R7609	104	886	421	56
0000058682	M2244	771	930	0	376
0000058682	C3322	565	0	596	13
0000058682	R7786	0	0	172	96
0000086362	M7765	0	190	557	0
0000086362	C8874	236	800	687	0

FIGURE 7-5: This macro ensures that only those rows that contain duplicate values are visible.

This macro enumerates through the cells in the target range, leveraging the For Each statement to activate each cell one at a time. It then uses the CountIf function to count the number of times the value in the active cell occurs in the range selected. If that number is 1, it hides the row in which the active cell resides. If that number is greater than 1, it formats the cell yellow and leaves the row visible.

```
Sub HighlightAndShowDupes()

'Step 1: Declare your variables.
  Dim MyRange As Range
  Dim MyCell As Range

'Step 2: Define the target range.
  Set MyRange = Selection

'Step 3: Start looping through the range.
  For Each MyCell In MyRange

'Step 4: Color the cell if there's a duplicate.
    If Not IsEmpty(MyCell.Value) Then

      If WorksheetFunction.CountIf(MyRange, MyCell.Value) > 1 Then
        MyCell.Interior.Color = vbYellow
        MyCell.EntireRow.Hidden = False
      Else
        MyCell.EntireRow.Hidden = True
      End If

    End If

'Step 5: Get the next cell in the range.
  Next MyCell

End Sub
```

1. Declares two Range object variables. One, called MyRange, holds the entire target range. The other, called MyCell, holds each cell in the range as you iterate through the cells one by one.
2. Stores the target range in the MyRange variable, the range that was selected on the spreadsheet in this example.

   You can easily set the MyRange variable to a specific range such as Range(“A1:Z100”). Also, if your target range is a named range, you could simply enter its name: Range(“MyNamedRange”).
3. Starts looping through each cell in the target range, assigning each cell to the control variable as it loops.

4. The IsEmpty function makes sure the cell is not empty. Do this so the macro won't automatically hide rows with no data in the target range.

   The WorksheetFunction object runs a CountIf function in VBA. This particular scenario counts the number of times the active cell value (MyCell.Value) is found in the given range (MyRange).

   If the CountIf expression evaluates to greater than 1, the interior color of the cell is changed and the EntireRow.Hidden property is set to False. This ensures the row is visible.

   If the CountIf expression does not evaluate to greater than 1, the macro jumps to the Else argument. Here the EntireRow.Hidden property is set to True. This ensures the row is not visible.

5. Loops back to get the next cell. After all cells in the target range are assigned, the macro ends.

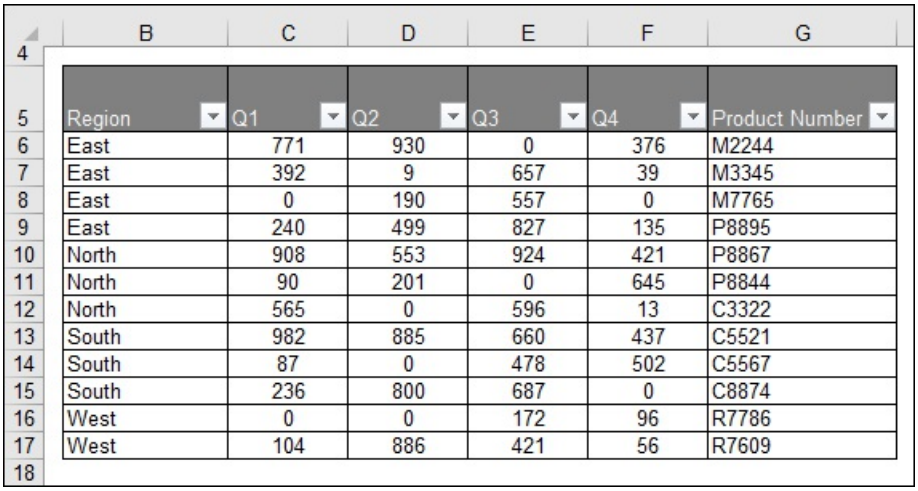


TIP Because the target range is defined as the current selection, you want to be sure to select the area where your data exists before running this code. You wouldn't want to select an entire column or the entire worksheet. Otherwise, any cell that contains data that is unique (not duplicated) triggers the hiding of the row. Alternatively, you can explicitly define the target range to ensure this is never a problem — such as Set MyRange = Range("A1:Z100").

To implement this macro, you can copy and paste it into a standard module, as shown in the first section of this chapter, "Installing Macros."

Selectively Hiding AutoFilter Drop-down Arrows

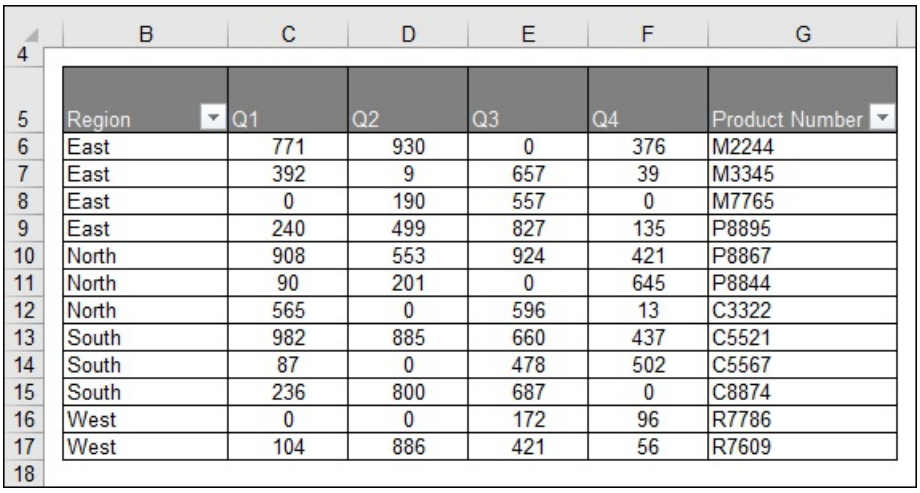
It goes without saying that the AutoFilter function in Excel is one of the most useful. Nothing else allows for faster on-the-spot filtering and analysis. The only problem is that the standard AutoFilter functionality applies drop-down arrows to every column in the chosen dataset (see Figure 7-6). This is all right in most situations, but what if you want to prevent your users from using the AutoFilter drop-down arrows on some of the columns in your data?



	B	C	D	E	F	G
4						
5	Region	Q1	Q2	Q3	Q4	Product Number
6	East	771	930	0	376	M2244
7	East	392	9	657	39	M3345
8	East	0	190	557	0	M7765
9	East	240	499	827	135	P8895
10	North	908	553	924	421	P8867
11	North	90	201	0	645	P8844
12	North	565	0	596	13	C3322
13	South	982	885	660	437	C5521
14	South	87	0	478	502	C5567
15	South	236	800	687	0	C8874
16	West	0	0	172	96	R7786
17	West	104	886	421	56	R7609
18						

FIGURE 7-6: The standard AutoFilter functionality adds drop-down arrows to all the columns in your data.

The good news is that with a little VBA, you can selectively hide AutoFilter drop-down arrows, as shown in Figure 7-7.



	B	C	D	E	F	G
4						
5	Region	Q1	Q2	Q3	Q4	Product Number
6	East	771	930	0	376	M2244
7	East	392	9	657	39	M3345
8	East	0	190	557	0	M7765
9	East	240	499	827	135	P8895
10	North	908	553	924	421	P8867
11	North	90	201	0	645	P8844
12	North	565	0	596	13	C3322
13	South	982	885	660	437	C5521
14	South	87	0	478	502	C5567
15	South	236	800	687	0	C8874
16	West	0	0	172	96	R7786
17	West	104	886	421	56	R7609
18						

FIGURE 7-7: With a little VBA, you can choose to hide certain AutoFilter drop-down arrows.

In VBA, you can use the AutoFilter object to turn on AutoFilters for a specific range. For example:

```
Range("B5:G5").AutoFilter
```

After an AutoFilter is applied, you can manipulate each of the columns in the AutoFilter by pointing to it. For example, you can perform some action on the third column in the AutoFilter, like this:

```
Range("B5:G5").AutoFilter Field:3
```

You can perform many actions on an AutoFilter field. In this scenario, you are interested in making the drop-down arrow on field 3 invisible. For this, you can use the VisibleDropDown parameter. Setting this parameter to False makes the drop-down arrow invisible:

```
Range("B5:G5").AutoFilter Field:3, VisibleDropDown:=False
```

Here is an example of a macro where you turn on AutoFilters, and then make only the first

and last drop-down arrows visible:

```
Sub HideAutoFilterArrows()

  With Range("B5:G5")
    .AutoFilter
    .AutoFilter Field:=1, VisibleDropDown:=True
    .AutoFilter Field:=2, VisibleDropDown:=False
    .AutoFilter Field:=3, VisibleDropDown:=False
    .AutoFilter Field:=4, VisibleDropDown:=False
    .AutoFilter Field:=5, VisibleDropDown:=False
    .AutoFilter Field:=6, VisibleDropDown:=True
  End With

End Sub
```



REMEMBER Not only are you pointing to a specific range, but you are also explicitly pointing to each field. When implementing this type of macro in your environment, alter the code to suit your particular data set.

To implement this macro, you can copy and paste it into a standard module. See the first section in this chapter, "Installing Macros," for more information.

Copying Filtered Rows to a New Workbook

Often, when you're working with an AutoFiltered set of data, you want to extract the filtered rows to a new workbook. Of course, you can manually copy the filtered rows, open a new workbook, paste the rows, and then format the newly pasted data so that all the columns fit. But if you are doing this frequently enough, you may want to have a macro to speed up the process.

This macro captures the AutoFilter range, opens a new workbook, and then pastes the data.

```
Sub CopyFilteredRows()

'Step 1: Check for AutoFilter – exit if none exists.
  If ActiveSheet.AutoFilterMode = False Then
    Exit Sub
  End If

'Step 2: Copy the AutoFiltered Range to new workbook.
  ActiveSheet.AutoFilter.Range.Copy
  Workbooks.Add.Worksheets(1).Paste

'Step 3: Size the columns to fit.
  Cells.EntireColumn.AutoFit

End Sub
```

1. The AutoFilterMode property checks whether the sheet even has AutoFilters applied. If not, the procedure ends.
2. Each AutoFilter object has a Range property. This Range property returns the rows to which the AutoFilter applies, meaning it returns only the rows shown in the filtered data set. The Copy method captures those rows, and then pastes the rows to a new workbook. Workbooks.Add.Worksheets(1) pastes the data into the first sheet of the newly created workbook.

3. Adjusts the column widths to autofit the data you just pasted.

To implement this macro, you can copy and paste it into a standard module. See the first section of this chapter, "Installing Macros," for details.

Showing Filtered Columns in the Status Bar

When you have a large table with many AutoFiltered columns, sometimes it's hard to tell which columns are filtered and which aren't. Of course, you could scroll through the columns, peering at each AutoFilter drop-down list for the telltale icon indicating the column is filtered, but that can get old quickly.

This macro helps by specifically listing all the filtered columns in the status bar. The status bar is the bar (shown in Figure 7-8) that runs across the bottom of the Excel window, below the sheet tabs.

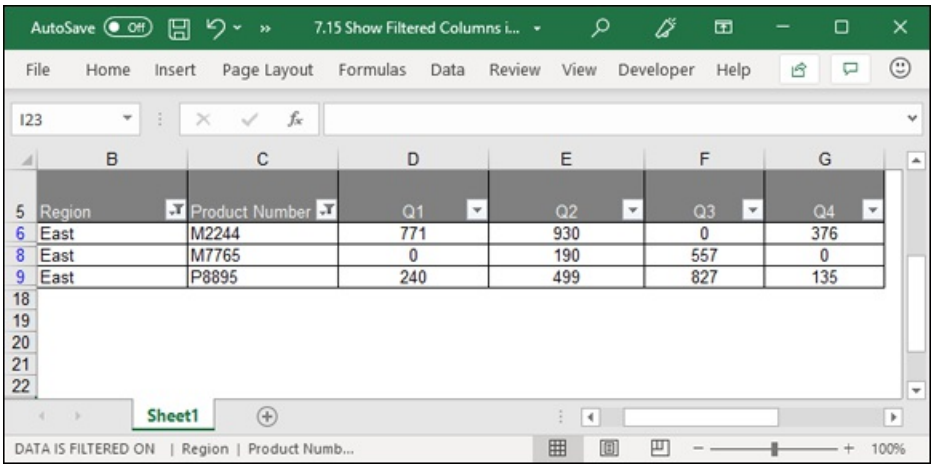


FIGURE 7-8: This macro lists all filtered columns in the status bar.

This macro loops through the fields in your AutoFiltered data set. As it loops, it checks to see whether each field is actually filtered. If so, it captures the field name in a text string. After looping through all the fields, it passes the final string to the StatusBar property.

```
Sub FilterInStatus()

'Step 1: Declare your variables.
  Dim AF As AutoFilter
  Dim TargetField As String
  Dim sOutput As String
  Dim i As Integer


'Step 2: Check if AutoFilter exists – if not exit.
  If ActiveSheet.AutoFilterMode = False Then
    Application.StatusBar = False
    Exit Sub
  End If

'Step 3: Set AutoFilter and start looping.
  Set AF = ActiveSheet.AutoFilter
  For i = 1 To AF.Filters.Count

'Step 4: Capture filtered field names.
    If AF.Filters(i).On Then
```

```
      TargetField = AF.Range.Cells(1, i).Value
      strOutput = sOutput & " | " & TargetField
     End If
  Next i


'Step 5: Display the filters if there are any.
  If Len(sOutput) = 0 Then
    Application.StatusBar = False
  Else
    Application.StatusBar = vbNullString
    Application.StatusBar = "DATA IS FILTERED ON " & sOutput
  End If

End Sub
```

1. Declares four variables. AF is an AutoFilter variable used to manipulate the AutoFilter object. TargetField is a string variable you use to hold the field names of any field that is filtered. The sOutput variable is a string variable used to build out the final text that goes into the status bar. Finally, the i variable serves as a simple counter, allowing you to iterate through the fields in your AutoFilter.
2. Checks the AutoFilterMode property to see whether a sheet even has AutoFilters applied. If not, the StatusBar property is set to False. This has the effect of clearing the Status bar and releasing control back to Excel. Then exit the procedure.
3. Sets the AF variable to the AutoFilter on the active sheet. The counter starts looping from 1 to the number of columns in the AutoFiltered range. The AutoFilter object keeps track of its columns with index numbers. Column 1 is index 1, column 2 is index 2, and so on. You can loop through each column in the AutoFilter by using the i variable as the index number.
4. Checks the status of the AF.Filters object for each filter. Pass the i variable to the Filters property to access the filter. If the AutoFilter for that column is filtered in any way, the On property is set to True.

   If the filter for the column is on, the value of the header cell is captured in the TargetField variable. Use the Range property of the AF AutoFilter object to get to the whole range. Then use the Cells property to access only the cell for the iteration you're on. Cells(1,1) captures the value in row 1, column 1. Cells(1,2) captures the value in row 1, column 2, and so on.

   The row argument to Cells is hard-coded to 1 and the column argument uses the i variable. As the macro iterates through the columns, it always captures the value in the first row of the auto-filtered table as the TargetField name (the first row is where the field name is).

   The TargetField name is joined to the sOutput variable with a pipe character to separate multiple fields. The sOutput variable keeps growing as field names are found and tacked on to the end of it.
5. Checks to make sure that the sOutput string holds a value. If sOutput is empty, it means the macro found no columns in your AutoFilter that were filtered. In this case, s the StatusBar property is set to False, releasing control back to Excel.

   If sOutput is not empty, the StatusBar property is set to equal some helper text along with the sOutput string.

You ideally want this macro to run each time a field is filtered. However, Excel does not have an OnAutoFilter event. The closest thing to that is the Worksheet_Calculate event. That being said, AutoFilters in themselves don't actually calculate anything, so you need to enter a "volatile" function on the sheet that contains your AutoFiltered data. A volatile function is one that forces a recalculation when any change is made on the worksheet.

In the sample files that come with this book, notice that the =Now() function is used. The Now function is a volatile function that returns a date and time. With this on the sheet, the worksheet is sure to recalculate each time the AutoFilter is changed.

Place the Now function anywhere on your sheet (by typing **=Now()** in any cell). Then copy and paste the macro into the Worksheet_Calculate event Code pane, as shown in Figure 7-9. (See the earlier section, "Installing Macros", for details.)

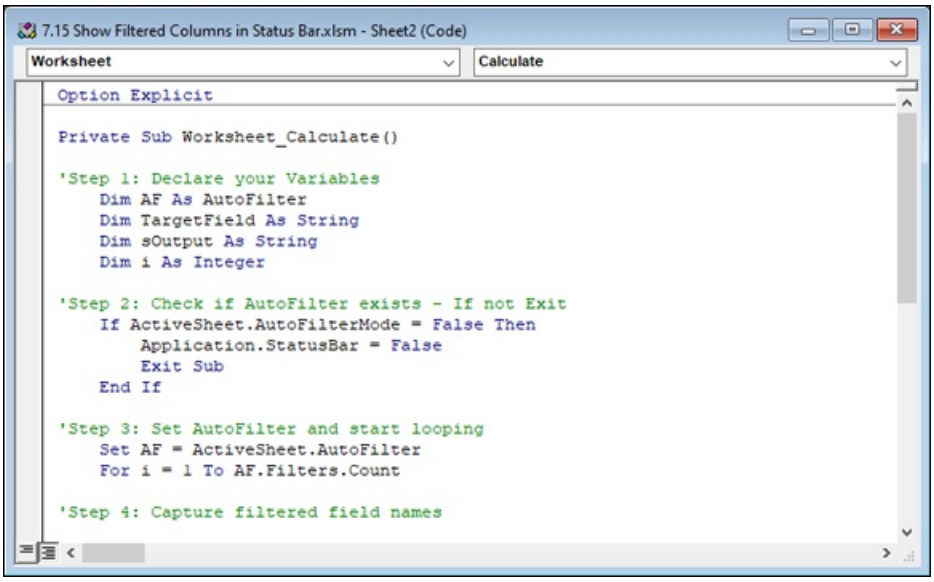


FIGURE 7-9: Type or paste your code in the Worksheet_Calculate event Code pane.



TIP To make the code run as smoothly as possible, consider adding these two pieces of code under the Worksheet_Calculate event:

```
Private Sub Worksheet_Deactivate()

  Application.StatusBar = False

End Sub

Private Sub Worksheet_Activate()

  Worksheet_Calculate

End Sub
```

Also, add this piece of code in the workbook BeforeClose event:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)

  Application.StatusBar = False
```

```
End Sub
```

The Worksheet_Deactivate event clears the status bar when you move to another sheet or workbook. This avoids confusion as you move between sheets.

The Worksheet_Activate event runs the macro in Worksheet_Calculate. This brings back the status bar indicators when you navigate back to the filtered sheet.

The Workbook_BeforeClose event clears the status bar when you close the workbook. This avoids confusion as you move between workbooks.

Part 4

Macro-Charging Reports and Emails

IN THIS PART …

Understand how macros can be used to automate your reporting processes.

Discover how to automate the more mundane aspects of building PivotTables.

See how macros can help work with the charts in your reports and dashboards.

Explore some of the techniques for sending emails from Excel.

Discover how to get data from external databases and files.

Chapter 8

Automating Common Reporting Tasks

IN THIS CHAPTER

- **Refreshing all PivotTables**
- **Creating a PivotTable inventory summary or a set of disconnected charts**
- **Adjusting all pivot data field titles**
- **Setting all data items to sums**
- **Setting the number format or custom sort for all data items**
- **Sorting all fields in alphabetical order**
- **Applying PivotTable and pivot field restrictions**
- **Automatically deleting PivotTable drilldown sheets**
- **Printing PivotTable for each report filter item or for all charts**
- **Creating new workbook for each report filter item**
- **Resizing or aligning charts**

If you're tasked with building dashboards and reports, PivotTables and charts are a daily part of your work life. You've probably never had the inclination to automate any aspect of these reporting tools with macros. But some aspects of that work lend themselves to a bit of automation. This chapter explores a handful of macros that can help you save time and gain efficiencies when working with PivotTables and charts.

Installing Macros

You can install the macros in this chapter in three places depending on what kind of macro it is and how available you want the macro to be. Event macros are put into the module of the object the event is tied to. Non-event macros go into a standard module and can be in the workbook in which they are meant to run or in a workbook called the Personal Macro Workbook if you want to be able to run them on any workbook.

Event macros

In VBA, objects have events. VBA "listens" for these events to happen and runs special event macros when it detects and event. For example, the workbook object has an Open event. When VBA detects that the workbook has been opened, it looks for a procedure called Workbook_Open in the ThisWorkbook module. If it's there, it runs that macro.

Most of the event macros that you'll use, and most of the examples in the book, are workbook or worksheet events. Workbook event macros go in the ThisWorkbook module and worksheet event macros go in the module for the sheet you want the event code to run on.

To install an event macro, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **In the Project Explorer, find your project/workbook name and click the plus sign next to it to see all the objects.**
3. **Double-click ThisWorkbook.**
4. **Select the event in the right drop-down list.**
5. **Type or paste the code in the module.**

Personal Macro Workbook

There is a special workbook called the Personal Macro Workbook named personal.xlsb. If you record a macro, you can choose this workbook as the destination of the recorded macro. The workbook doesn't exist until you first record a macro into it. After that, it lives in a folder and Excel loads that workbook every time it starts.

The benefit of a workbook that opens every time Excel starts is that all the macros in that workbook are always available to you. If you have a macro that works on the active sheet or whatever range is currently selected, that's a good candidate for the Personal Macro Workbook.

To install a macro in the Personal Macro Workbook, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click personal.xlb in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**



REMEMBER If you don't see personal.xlsb in the Project Explorer, record a macro to the Personal Macro Workbook and Excel will create it for you.

Standard macros

All the other macro examples in this book are installed in a specific workbook in a standard module. If it's not an event macro and it's not a macro you want in the Personal Macro Workbook, then install it using the following steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click the project/workbook name in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**

Refreshing All PivotTables in a Workbook

It's not uncommon to have multiple PivotTables in the same workbook. Many times, these PivotTables link to data that changes, requiring a refresh of the PivotTables. If you find that you need to refresh your PivotTables en masse, you can use this macro to refresh all PivotTables on demand.

It's important to know that each PivotTable object is a child of the worksheet it sits in. The macro has to loop through the worksheets in a workbook first, and then loop through the PivotTables in each worksheet. This macro does just that — loops through the worksheets, and then loops through the PivotTables. On each loop, the macro refreshes the PivotTable.

```
Sub RefreshAllPTs()

'Step 1: Declare your variables.
  Dim ws As Worksheet
  Dim pt As PivotTable

'Step 2: Loop through each sheet in workbook.
  For Each ws In ThisWorkbook.Worksheets

'Step 3: Loop through each PivotTable.
    For Each pt In ws.PivotTables
      pt.RefreshTable
    Next pt
  Next ws

End Sub
```

1. Declares an object variable called w, and creates a memory container for each worksheet you loop through. An object variable called pt is also declared, which holds each PivotTable you loop through.
2. Starts looping through each worksheet in the workbook. ThisWorkbook is used rather than ActiveWorkbook to refer to the workbook that the code is contained in as opposed to the workbook that is currently active.
3. Loops through each PivotTable in the worksheet currently assigned to the ws variable and uses the RefreshTable method. After all PivotTables have been refreshed, the next sheet is assigned. After all sheets have been assigned, the macro ends.

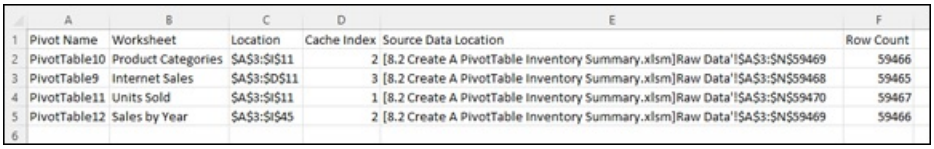


TIP As an alternative method for refreshing all PivotTables in the workbook, you can use ThisWorkbook.RefreshAll. This refreshes all the PivotTables in the workbook. However, it also refreshes all query tables also. So if you have data tables connected to an external source or the web, these are also affected by the RefreshAll method. If this isn't a concern, you can simply enter ThisWorkbook.RefreshAll into a standard module.

To implement this macro, you can copy and paste it into a standard module. See the previous section for more information.

Creating a List of PivotTables

When your workbook contains multiple PivotTables, it can be helpful to have a list of them (similar to the one shown in Figure 8-1) that outlines basic details about the PivotTables. With this type of summary, you can quickly see important information such as the location of each PivotTable, the location of each PivotTable's source data, and the pivot cache index each PivotTable is using.



	A	B	C	D	E	F
1	Pivot Name	Worksheet	Location	Cache Index	Source Data Location	Row Count
2	PivotTable10	Product Categories	$A$3:$I$11	2	[8.2 Create A PivotTable Inventory Summary.xlsm]Raw Data'!$A$3:$N$59469	59466
3	PivotTable9	Internet Sales	$A$3:$D$11	3	[8.2 Create A PivotTable Inventory Summary.xlsm]Raw Data'!$A$3:$N$59468	59465
4	PivotTable11	Units Sold	$A$3:$I$11	1	[8.2 Create A PivotTable Inventory Summary.xlsm]Raw Data'!$A$3:$N$59470	59467
5	PivotTable12	Sales by Year	$A$3:$I$45	2	[8.2 Create A PivotTable Inventory Summary.xlsm]Raw Data'!$A$3:$N$59469	59466
6						

FIGURE 8-1: A PivotTable inventory summary.

The following macro outputs such a summary.

When you create a PivotTable object variable, you expose all the PivotTable's properties — such as its name, location, cache index, and so on. In this macro, you loop through each PivotTable in the workbook and extract specific properties into a new worksheet.

Because each PivotTable object is a child of the worksheet it's in, you have to loop through the worksheets in a workbook first, and then loop through the PivotTables in each worksheet.

Take a moment to walk through the steps of this macro in detail.

```
Sub CreatePTList()

'Step 1: Declare your variables.
  Dim ws As Worksheet
  Dim pt As PivotTable
  Dim Cell As Range

'Step 2: Add a new sheet with column headers.
  Worksheets.Add
  Range("A1:F1") = Array("Pivot Name", "Worksheet", _
    "Location", "Cache Index", "Source Data Location", _
    "Row Count")

'Step 3: Start cursor at cell A2 setting the anchor here.
  Set Cell = ActiveSheet.Range("A2")

'Step 4: Loop through each sheet in workbook.
  For Each ws In ThisWorkbook.Worksheets

'Step 5: Loop through each PivotTable.
    For Each pt In ws.PivotTables
      Cell.Offset(0, 0).Value = pt.Name
      Cell.Offset(0, 1).Value = pt.Parent.Name
      Cell.Offset(0, 2).Value = pt.TableRange2.Address
      Cell.Offset(0, 3).Value = pt.CacheIndex
      Cell.Offset(0, 4).Value = Application.ConvertFormula _
       (pt.PivotCache.SourceData, xlR1C1, xlA1)
      Cell.Offset(0, 5).Value = pt.PivotCache.RecordCount

'Step 6: Move cursor down one row and set a new anchor.
      Set Cell = Cell.Offset(1, 0)

'Step 7: Work through all PivotTables and worksheets.
    Next pt
  Next ws

'Step 8: Size columns to fit.
  ActiveSheet.Cells.EntireColumn.AutoFit

End Sub
```

1. Declares object variables ws and pt to act as control variables for looping through the worksheets and PivotTables. A range variable called Cell is then declared to keep track of which row you're on when creating the list.
2. Creates a new worksheet and adds column headings that range from A1 to F1. Note that you can add column headings using a simple array that contains your header labels. This new worksheet remains your active sheet from here on out.
3. Assigns cell A2 to the Cell variable. This is the starting position for your list and what Cell points to changes for every iteration of the loop.
4. Starts looping through each worksheet in the workbook.
5. Starts looping through each PivotTable in the currently assigned worksheet. The appropriate property from each PivotTable is extracted and filled in the table based on what Cell is pointing to (see Step 3).

   Use six PivotTable properties: Name, Parent.Name, TableRange2.Address, CacheIndex, PivotCache.SourceData, and PivotCache.RecordCount.

   - The Name property returns the name of the PivotTable.
   - The Parent.Name property gives the sheet where the PivotTable resides.
   - The TableRange2.Address property returns where on the worksheet the PivotTable is.
   - The CacheIndex property returns the index number of the PivotCache for the PivotTable. A PivotCache stores all the data for a PivotTable. When you create a new PivotTable, Excel takes a snapshot of the source data and creates a PivotCache. Each time you refresh a PivotTable, Excel goes back to the source data and takes another snapshot and updates the PivotCache. Each PivotCache has a SourceData property that identifies the location of the data used to create the pivot cache.
   - The PivotCache.SourceData property tells you which range is called upon when you refresh the PivotTable. By default, the PivotCache.SourceData property returns range addresses in the R1C1 reference style — for example: 'Raw Data'!R1C1:R5000C14. The address is converted to the A1 reference style using the Application.ConvertFormula method.
   - The PivotCache.RecordCount property outputs the count of records found in the PivotCache.

   Each of these properties is assigned to a cell on the newly created worksheet. The Offset property is used to write each result to a different cell in the same row. The Offset property takes two arguments: RowOffset and ColumnOffset. In this example, RowOffset is always zero because you want to write the results to the same row. ColumnOffset is used to refer to cells to the right of the Cell variable. For example, Cell.Offset(0,3) refers to the cell three cells to the right of Cell. RowOffset and ColumnOffset can also be negative to refer to rows above or columns to the left of the anchor cell.
6. Before you move to the next PivotTable, Cell is assigned to the cell one row down from

its current assignment, starting a new row for each PivotTable.

7. Loops back to iterate through all PivotTables and all worksheets. After all PivotTables have been assigned, you move to the next sheet. After all sheets have been assigned, the macro moves to the last step.
8. Sizes the columns to fit the data.

To implement this macro, you can copy and paste it into a standard module, as outlined in the first section of this chapter, "Installing Macros."

Adjusting All Pivot Data Field Titles

When you create a PivotTable, Excel tries to help you by prefacing each data field header with Sum of, Count of, or whichever operation you use. If you want clean titles that match your data source as closely as possible, you can type your own titles manually. Or you can use this macro to fix them all in one go.

Ideally, the name of the each data item matches the field name from your source data set (the original source data used to create the PivotTable). Unfortunately, PivotTables won't allow you to name a data field the exact name as the source data field. The workaround for this is to add a space to the end of the field name. Excel considers the field name (with a space) to be different from the source data field name, so it allows it. Cosmetically, the readers of your spreadsheet don't notice the space after the name.

This macro uses this workaround to rename your data fields. It loops through each data field in the PivotTable, and then resets each header to match its respective field in the source data plus a space character.

```
Sub AdjustTitles()

'Step 1: Declare your variables.
  Dim pt As PivotTable
  Dim pf As PivotField

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside a PivotTable."
    Exit Sub
  End If

'Step 4: Loop through all pivot fields adjust titles.
  For Each pf In pt.DataFields
    pf.Caption = pf.SourceName & Chr(160)
  Next pf

End Sub
```

1. Declares an object variable called pt to hold the PivotTable and one called pf to serve as a control variable for looping through the pivot fields.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a PivotTable for this macro to run. The Range object has a PivotTable property that returns

the object you want.



TIP Use the On Error Resume Next statement to ignore the error you would get if the active cell isn't contained within a PivotTable. The next step deals with errors.

3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, a message box is displayed for the user and the procedure exits.
4. The PivotTable is successfully pointed to and loops through each data field. Each time a new pivot field is assigned, the field name sets the Caption property to match the field's SourceName plus a non-breaking space.



REMEMBER Every character has an underlying ASCII code, similar to a serial number. For example, the lowercase letter *a* has an ASCII code of 97. The lowercase letter *c* has an ASCII code of 99. Likewise, invisible characters such as the space have a code. You can use invisible characters in your macro by passing their code through the CHR function.

After the name has been changed, the macro moves to the next data field. After all the data fields have been evaluated, the macro ends.

To implement this macro, you can copy and paste it into a standard module. (See the "Installing Macros" section at the beginning of this section.)

Setting All Data Items to Sum

When creating a PivotTable, Excel, by default, summarizes your data by either counting or summing the items. The logic Excel uses to decide whether to sum or count the fields you add to your PivotTable is fairly simple. If all the cells in a column contain numeric data, Excel chooses to Sum. If the field you are adding contains a blank or text, Excel chooses Count.

Although this seems to make sense, in many instances, a pivot field that should be summed legitimately contains blanks. In these cases, you're forced to manually change the calculation type from Count back to Sum. That's if you're paying attention! It's not uncommon to miss the fact that a pivot field is being counted rather than summed.

The macro in this section aims to help by automatically setting each data item's calculation type to Sum.

This macro loops through each data field in the PivotTable and changes the Function property to xlSum. You can alter this macro to use any one of the calculation choices: xlCount, xlAverage, xlMin, xlMax, and so on. When you type **pf.Function =** into the Code pane, you see a drop-down list showing you all your choices (see Figure 8-2).

```
Sub SetToSum()
```

```
'Step 1: Declare your variables.
  Dim pt As PivotTable
  Dim pf As PivotField

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside a PivotTable."
    Exit Sub
  End If

'Step 4: Loop through all pivot fields apply SUM.
  For Each pf In pt.DataFields
    pf.Function = xlSum
  Next pf

End Sub
```

1. Declares an object variable called pt to hold the PivotTable and one called pf to serve as a control variable for looping through the PivotFields.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a PivotTable for this macro to run. The Range object has a PivotTable property that returns the object you want.

   Use the On Error Resume Next statement to ignore the error you would get if the active cell isn't contained within a PivotTable. The next step deals with errors.
3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, message box alerts the user and the procedure closes.
4. The PivotTable is successfully pointed to and starts looping through each data field. Each time a new pivot field is assigned, the Function property sets the calculation used by the field, sum in this case.

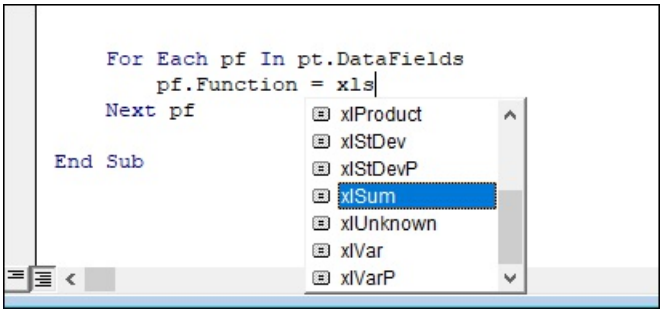


FIGURE 8-2: Excel shows the enumeration choices.

After the name has been changed, you move to the next data field. After all the data fields have been evaluated, the macro ends.

To implement this macro, you can copy and paste it into a standard module. The first section in this chapter, "Installing Macros," outlines these steps.

Applying Number Formatting for All Data Items

A PivotTable does not inherently store number formatting in its PivotCache. Formatting takes up memory; to be as lean as possible, the PivotCache contains only data. Unfortunately, this results in the need to apply number formatting to every field you add to a PivotTable. This takes from eight to ten clicks of the mouse for every data field you add. When you have PivotTables that contain five or more data fields, you're talking about more than 40 clicks of the mouse!

Ideally, a PivotTable should be able to look back at its source data and adopt the number formatting from the fields there. The macro outlined in this section is designed to do just that. It recognizes the number formatting in the PivotTable's source data and applies the appropriate formatting to each field automatically.

Before running this code, you want to make sure that

- The source data for your PivotTable is accessible. The macro needs to see it to capture the correct number formatting.
- The source data is appropriately formatted. Money fields are formatted as currency, value fields are formatted as numbers, and so on.

This macro uses the PivotTable SourceData property to find the location of the source data. It then loops through each column in the source, capturing the header name and the number format of the first value under each column. After it has that information, the macro determines whether any of the data fields match the evaluated column. If it finds a match, the number formatting is applied to that data field.

```
Sub ApplyNumberFormats()

'Step 1: Declare your variables.
  Dim pt As PivotTable
  Dim pf As PivotField
  Dim SrcRange As Range
  Dim sFormat As String
  Dim sLabel As String
  Dim i As Long

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside of a PivotTable."
    Exit Sub
  End If

'Step 4: Capture the source range.
  Set SrcRange = _
  Range(Application.ConvertFormula(pt.SourceData, xlR1C1, xlA1))


'Step 5: Start looping through the columns in source range.
  For i = 1 To SrcRange.Columns.Count

'Step 6: Store the source column name and number format
```

```
    sLabel = SrcRange.Cells(1, i).Value
    sFormat = SrcRange.Cells(2, i).NumberFormat

'Step 7: Loop through the fields PivotTable data area.
    For Each pf In pt.DataFields

'Step 8: Check for match on SourceName then apply format.
      If pf.SourceName = sLabel Then
        pf.NumberFormat = sFormat
      End If
    Next pf
  Next i

End Sub
```

1. Declares six variables:
    - The pt variable holds the PivotTable.
    - The pf holds the control variable for looping through the data fields.
    - The SrcRange variable holds the data range for the source data.
    - The sFormat and sLabel variables are both text string variables used to hold the source column label and number formatting respectively.
    - The i variable serves as a control variable for looping through the columns of the source data range.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a PivotTable for this macro to run. The Range object has a PivotTable property that returns the object you want.
3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, a message box alerts the user and the procedure closes.
4. A PivotTable is successfully pointed to and the PivotTable's source data range is assigned to the SrcRange variable.



TIP All PivotTables have a SourceData property that points to the address of its source. Unfortunately, the address is stored in the R1C1 reference style so you need to convert it to A1 sytle using the ConvertFormula method.

5. After the range is assigned, the columns in the source range are looped through. The i variable is used as an index number for the columns in the source range. The index number starts at 1 and ends at the number of columns in the source range.
6. As the macro loops through the columns in the source range, the column header label and the number format are captured.



REMEMBER The Cells property offers an extremely handy way of referring to ranges through code. It requires only relative row and column positions as parameters. Cells (1,1)

translates to row 1, column 1 (or the header row of the first column). Cells (2, 1) translates to row 2, column 1 (or the first value in the first column).

sLabel is filled by the header label taken from row 1 of the selected column. sFormat is filled with the number formatting from row 2 of the selected column.

7. Starts looping through the data fields in the PivotTable.
8. Compares each data field to see whether its source matches the name in sLabel. If it does, that means the number formatting captured in sFormat belongs to that data field.
9. After all data fields have been evaluated, i is incremented to the next column in the source range. After all columns have been evaluated, the macro ends.

To implement this macro, you can copy and paste it into a standard module. The first section in this chapter, "Installing Macros," shows you how.

Sorting All Fields in Alphabetical Order

If you frequently add data to your PivotTables, you may notice that new data doesn't automatically fall into the sort order of the existing pivot data. Instead, it gets tacked to the bottom of the existing data. This means that your drop-down lists show all new data at the very bottom, whereas existing data is sorted alphabetically.

This macro works to reset the sorting on all data fields, ensuring that any new data snaps into place. The idea is to run it each time you refresh your PivotTable. In the code, you loop through each data field in the PivotTable, sorting each one as you go.

```
Sub SortAlpha()

'Step 1: Declare your variables.
  Dim pt As PivotTable
  Dim pf As PivotField

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside a PivotTable."
    Exit Sub
  End If

'Step 4: Loop through all pivot fields and sort.
  For Each pf In pt.PivotFields
    pf.AutoSort xlAscending, pf.Name
  Next pf

End Sub
```

1. Declares an object variable called pt to hold the PivotTable and one called pf to serve as a control variable for looping through the PivotFields.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a PivotTable for this macro to run. The Range object has a PivotTable property that returns the object you want.

   Use the On Error Resume Next statement to ignore the error you would get if the active

cell isn't contained within a PivotTable. The next step deals with errors.

3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, a message box alerts the user and the procedure exits.
4. With a PivotTable successfully pointed to, looping through each data field is started. Each time a new pivot field is assigned, re-sort it by calling the AutoSort method.

To implement this macro, you can copy and paste it into a standard module. See the first section of this chapter, "Installing Macros," for the details.

Applying a Custom Sort to Data Items

On occasion, you may need to apply a custom sort to the data items in your PivotTable. For example, if you work for a company in California, your organization may want the West region to come before the North and South. In these types of situations, neither the standard ascending nor descending sort order work.

You can automate the custom sorting of your fields by using the Position property of the PivotItems object. With the Position property, you can assign a position number that specifies the order in which you want to see each pivot item.

In this example code, you first point to the Region pivot field in the Pvt1 PivotTable. Then you list each item along with the position number indicating the customer sort order you need.

```
Sub ApplyCustomSort()

  With Sheets("Sheet1").PivotTables("Pvt1").PivotFields _
    ("Region ")
    .PivotItems("West").Position = 1
    .PivotItems("North").Position = 2
    .PivotItems("South").Position = 3
  End With

End Sub
```



TIP The other solution is to set up a custom sort list. Custom sort lists are defined lists stored in your instance of Excel. To create a custom sort list, go to the Excel Options dialog box and choose Edit Custom Lists. Here, you can type West, North, and South in the List Entries box and click the Add button. After setting up a custom list, you can select a data item in the target field (in this case, Region) and then choose Data ⇒ Sort & Filter ⇒ Sort. In the Sort dialog box that opens, you can select Custom List from the Order drop-down list.

As brilliant as this option is, custom lists do not travel with your workbook. So a macro helps in cases where it's impractical to expect your audience to set up their own custom lists.

You can implement this kind of macro in a standard module, as outlined in the first section of this chapter, "Installing Macros."

Applying PivotTable Restrictions

If you send PivotTables to clients, co-workers, managers, and other groups of people, you probably need to restrict the types of actions users can take on the PivotTable reports. The macro outlined in this section demonstrates some of the protection settings available via VBA.

The PivotTable object exposes several properties that allow you (the developer) to restrict different features and components of a PivotTable:

- **EnableWizard:** Setting this property to False disables the PivotTable Tools context menu that normally activates when clicking inside a PivotTable.
- **EnableDrilldown:** Setting this property to False prevents users from getting to detailed data by double-clicking a data field.
- **EnableFieldList:** Setting this property to False prevents users from activating the field list or moving pivot fields around.
- **EnableFieldDialog:** Setting this property to False disables the users' ability to alter the pivot field via the Value Field Settings dialog box.
- **PivotCache.EnableRefresh:** Setting this property to False disables the ability to refresh the PivotTable.

You can set any or all of these properties independently to either True or False. In this macro, you apply all the restrictions to the target PivotTable.

```
Sub RestrictPT()

'Step 1: Declare your variables.
  Dim pt As PivotTable

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside a PivotTable."
    Exit Sub
  End If

'Step 4: Apply PivotTable restrictions.
  With pt
    .EnableWizard = False
    .EnableDrilldown = False
    .EnableFieldList = False
    .EnableFieldDialog = False
    .PivotCache.EnableRefresh = False
  End With

End Sub
```

1. Declares pt to hold a reference to the PivotTable.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a PivotTable for this macro to run. The Range object has a PivotTable property that returns the object you want.

Use the On Error Resume Next statement to ignore the error you would get if the active cell isn't contained within a PivotTable. The next step deals with errors.

3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, a message box alerts the user and the procedure exits.
4. Applys all PivotTable restrictions.

You can implement this kind of macro in a standard module, as outlined in the first section of this chapter, "Installing Macros."

Applying Pivot Field Restrictions

Like PivotTable restrictions, pivot field restrictions enable you to restrict the types of actions users can take on the pivot fields in a PivotTable. The macro outlined in this section demonstrates some of the protection settings available via VBA.

The PivotField object exposes several properties that allow you (the developer) to restrict different features and components of a PivotTable:

- **DragToPage:** Setting this property to False prevents the users from dragging any pivot field into the Report Filter area of the PivotTable.
- **DragToRow:** Setting this property to False prevents the users from dragging any pivot field into the Row area of the PivotTable.
- **DragToColumn:** Setting this property to False prevents the users from dragging any pivot field into the Column area of the PivotTable.
- **DragToData:** Setting this property to False prevents the users from dragging any pivot field into the Data area of the PivotTable.
- **DragToHide:** Setting this property to False prevents the users from dragging pivot fields off the PivotTable. It also prevents the use of the right-click menu to hide or remove pivot fields.
- **EnableItemSelection:** Setting this property to False disables the drop-down lists on each pivot field.

You can set any or all of these properties independently to either True or False. In this macro, you apply all the restrictions to the target PivotTable.

```
Sub Macro1()

'Step 1: Declare your variables.
  Dim pt As PivotTable
  Dim pf As PivotField

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
```

```
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside a PivotTable."
    Exit Sub
  End If

'Step 4: Apply pivot field restrictions.
  For Each pf In pt.PivotFields
    pf.EnableItemSelection = False
    pf.DragToPage = False
    pf.DragToRow = False
    pf.DragToColumn = False
    pf.DragToData = False
    pf.DragToHide = False
  Next pf

End Sub
```

1. Declares an object variable called pt to hold the PivotTable and one called pf to serve as a control variable for looping through the pivot fields.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a PivotTable for this macro to run. The Range object has a PivotTable property that returns the object you want.

   Use the On Error Resume Next statement to ignore the error you would get if the active cell isn't contained within a PivotTable. The next step covers errors.
3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, a message box alerts the user and the procedure exits.
4. Starts looping through each pivot field. Each time a new pivot field is assigned, all the pivot field restrictions are applied.

You can implement this kind of macro in a standard module. See the "Installing Macros" section at the beginning of this chapter for details.

Automatically Deleting PivotTable Drill-Down Sheets

One of the coolest features of a PivotTable is that it gives you the ability to double-click a number and drill into the details. The details are output to a new sheet that you can review. In most cases, you don't want to keep these sheets. In fact, they often become a nuisance, forcing you to take the time to clean them up by deleting them.

This is especially a problem when you distribute PivotTable reports to users who frequently drill into details. There is no guarantee they will remember to clean up the drill-down sheets. Although these sheets probably won't cause issues, they can clutter up the workbook.

Here is a technique you can implement to have your workbook automatically remove these drill-down sheets.

The basic premise of this macro is actually very simple. When the user clicks for details, outputting a drill-down sheet, the macro simply renames the output sheet so that the first ten

characters are PivotDrill. Then before the workbook closes, the macro finds any sheet that starts with PivotDrill and deletes it.

The implementation does get a bit tricky because you essentially have to have two pieces of code. One piece goes in the Worksheet_BeforeDoubleClick event; the other piece goes into the Workbook_BeforeClose event.

```
Private Sub Worksheet_BeforeDoubleClick(ByVal Target As Range, Cancel As Boolean)

'Step 1: Declare your variables.
  Dim pt As String

'Step 2: Exit if double-click did not occur on a PivotTable.
  On Error Resume Next
  If IsEmpty(Target) And ActiveCell.PivotField.Name <> "" Then
    Cancel = True
    Exit Sub
  End If

'Step 3: Set the PivotTable object.
  pt = ActiveSheet.Range(ActiveCell.Address).PivotTable

'Step 4: If drilldowns are enabled, drill down.
  If ActiveSheet.PivotTables(pt).EnableDrilldown Then
    Selection.ShowDetail = True

    ActiveSheet.Name = _
      Replace(ActiveSheet.Name, "Sheet", "PivotDrill")
  End If

End Sub
```

1. Creates the pt string variable to hold the PivotTable's name.
2. Checks the double-clicked cell. If the cell is not associated with any PivotTable, the double-click event is cancelled.
3. Fills the pt variable with the PivotTable if the PivotTable is associated with a cell.
4. Checks the EnableDrillDown property. If it is enabled, the ShowDetail method is called. This outputs the drill-down details to a new worksheet.

   The output sheet is renamed so that the first ten characters are PivotDrill. This is accomplished with the Replace function, which replaces certain text in an expression with other text. In this case, you're replacing the word Sheet with PivotDrill.

   Sheet1 becomes PivotDrill1, Sheet12 becomes PivotDrill12, and so on.

Next, the Worksheet_BeforeDoubleClick event is set up. As the name suggests, this code runs when the workbook closes:

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)

'Step 1: Declare your variables.
  Dim ws As Worksheet


'Step 2: Loop through worksheets.
  For Each ws In ThisWorkbook.Worksheets

'Step 3: Delete any sheet that starts with PivotDrill.
    If Left(ws.Name, 10) = "PivotDrill" Then
      Application.DisplayAlerts = False
      ws.Delete
      Application.DisplayAlerts = True
    End If
```

```
  Next ws

End Sub
```

1. Declares the ws Worksheet variable to hold worksheet objects as you loop through the workbook.
2. Starts looping through the worksheets in this workbook.
3. Determines whether the left ten characters of that sheet name are PivotDrill. If so, the worksheet is deleted while alerts are temporarily turned off. After all the sheets have been evaluated, all drill-down sheets have been cleaned up and the macro ends.

To implement the first part of the macro, you need to copy and paste it into the sheet's module, as shown in Figure 8-3. (See the first section of this chapter, "Installing Macros.") Placing the macro here allows it to run each time you double-click the sheet.

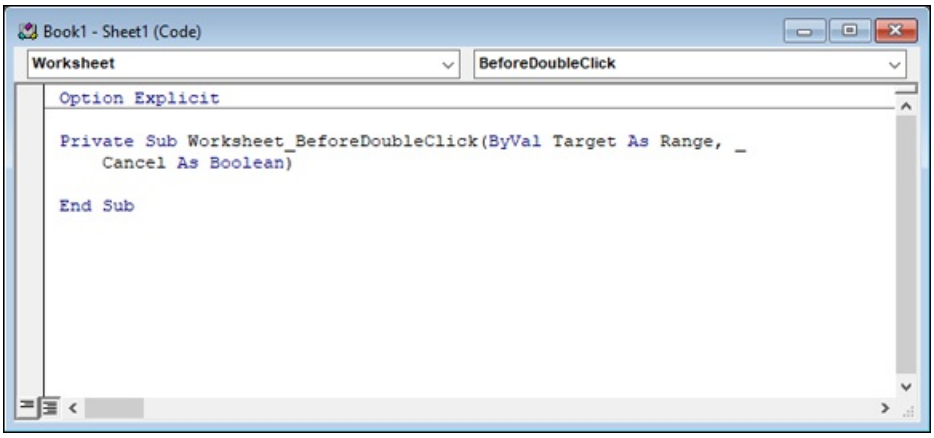


FIGURE 8-3: Type or paste your code in the Worksheet_BeforeDoubleClick event Code pane.

For the second macro, copy and paste it into the ThisWorkbook module, as shown in Figure 8-4. (See the section "Installing Macros" at the beginning of this chapter.) Placing the macro here allows it to run each time you try to close the workbook.

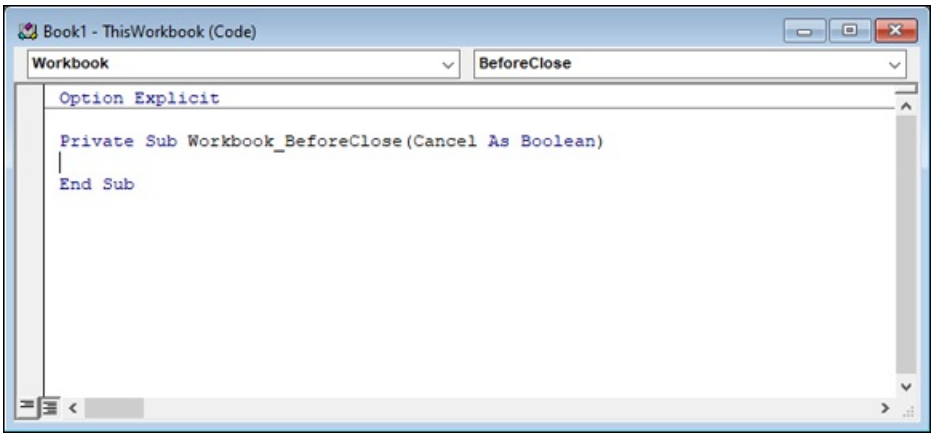


FIGURE 8-4: Enter or paste your code in the Workbook_BeforeClose event Code pane.

Printing a PivotTable for Each Report Filter Item

PivotTables provide an excellent mechanism to parse large data sets into printable files. You can build a PivotTable report, complete with aggregations and analysis, and then place a field (such as Region) into the report filter. With the report filter, you can select each data item at a time, and then print the PivotTable report.

The macro in this section demonstrates how to automatically iterate through all the values in a report filter and print.

In the Excel object model, the Report Filter drop-down list is known as the PageField. To print a PivotTable for each data item in a report filter, you need to loop through the PivotItems collection of the PageField object. As you loop, you dynamically change the selection in the report filter, and then use the ActiveSheet.PrintOut method to print the target range.

```
Sub PrintPTReport()

'Step 1: Declare your variables.
  Dim pt As PivotTable
  Dim pf As PivotField
  Dim pi As PivotItem

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside a PivotTable."
    Exit Sub
  End If

'Step 4: Exit if more than one page field.
  If pt.PageFields.Count > 1 Then
    MsgBox "Too many Report Filter Fields. Limit 1."
    Exit Sub
  End If

'Step 5: Start looping through the page field and its pivot items.
  For Each pf In pt.PageFields
    For Each pi In pf.PivotItems

'Step 6: Change the selection in the report filter.
      pt.PivotFields(pf.Name).CurrentPage = pi.Name

'Step 7: Set Print Area and print.
      ActiveSheet.PageSetup.PrintArea = pt.TableRange2.Address
      ActiveSheet.PrintOut Copies:=1

'Step 8: Get the next page field item.
    Next pi
  Next pf

End Sub
```

1. Declares an object variable called pt to hold the PivotTable, one called pf to serve as a control variable for looping through the PivotFields, and one called pi to serve as a control variable for looping through the PivotItems.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a

PivotTable for this macro to run. The Range object has a PivotTable property that returns the object you want.

Use the On Error Resume Next statement to ignore the error you would get if the active cell isn't contained within a PivotTable. The next step deals with errors.

3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, a message box alerts the user and the procedure exits.
4. Determines whether there is more than one report filter field. (If the count of PageFields is greater than 1, there is more than one report filter.) This check is necessary for a simple reason: If you have multiple report filter fields, you might wind up printing hundreds of pages. The macro stops with a message box if the PivotTable contains more than one report filter field. You can remove this limitation simply by deleting or commenting out Step 4 in the macro.
5. Starts looping through each PageField and PivotItem. The outer loop tells Excel to iterate through all the report filters. The inner loop tells Excel to loop through all the pivot items in the report filter that currently is assigned.
6. For each pivot item, the report filter's CurrentPage property is changed to the currently assigned PivotItem's name. This alters the PivotTable report to match the pivot item.
7. Prints the active sheet and then moves to the next pivot item. After you have looped through all pivot items in the report filter, the macro moves to the next PageField. After all PageFields have been assigned, the macro ends.

You can implement this kind of macro in a standard module. For installing details, see the first section in this chapter, “Installing Macros.”

Creating a New Workbook for Each Report Filter Item

PivotTables provide an excellent mechanism to parse large data sets into separate files. You can build a PivotTable report, complete with aggregations and analysis, and then place a field (such as Region) into the report filter. With the report filter, you can select each data item at a time, and then export the PivotTable data to a new workbook.

The macro in this section demonstrates how to automatically iterate through all the values in a report filter and export each to a new workbook.

In the Excel object model, the Report Filter drop-down list is known as the PageField. To copy a PivotTable report for each data item in a report filter, the macro needs to loop through the PivotItems collection of the PageField object. As the macro loops, it must dynamically change the selection in the report filter, and then export the PivotTable report to a new workbook.

```
Sub CreateNewWbs()

'Step 1: Declare your variables.
  Dim pt As PivotTable
  Dim pf As PivotField
  Dim pi As PivotItem

'Step 2: Point to the PivotTable in the active cell.
  On Error Resume Next
  Set pt = ActiveCell.PivotTable

'Step 3: Exit if active cell is not in a PivotTable.
  If pt Is Nothing Then
    MsgBox "You must place your cursor inside a PivotTable."
    Exit Sub
  End If

'Step 4: Exit if more than one page field.
  If pt.PageFields.Count > 1 Then
    MsgBox "Too many Report Filter Fields. Limit 1."
    Exit Sub
  End If

'Step 5: Start looping through the page field and its pivot items.
  For Each pf In pt.PageFields
    For Each pi In pf.PivotItems

'Step 6: Change the selection in the report filter.
      pt.PivotFields(pf.Name).CurrentPage = pi.Name

'Step 7: Copy the data area to a new workbook.
      pt.TableRange1.Copy

      Workbooks.Add.Worksheets(1).Paste

      Application.DisplayAlerts = False
        ActiveWorkbook.SaveAs _
        Filename:= ThisWorkbook.Path & "\" _
          & pi.Name & ".xlsx"
        ActiveWorkbook.Close
      Application.DisplayAlerts = True

'Step 8: Get the next page field item.
    Next pi
  Next pf
```

```
End Sub
```

1. Declares an object variable called pt to hold the PivotTable, one called pf to serve as a control variable for looping through the PivotFields, and one called pi to serve as a control variable for looping through the PivotItems.
2. Assigns pt to whatever PivotTable the active cell is in. The active cell must be inside a PivotTable for this macro to run. The Range object has a PivotTable property that returns the object you want.

   Use the On Error Resume Next statement to ignore the error you would get if the active cell isn't contained within a PivotTable. The next step deals with errors.
3. Checks to see whether the pt variable is filled with a PivotTable object. If the pt variable is set to Nothing, the active cell was not in a PivotTable, thus no PivotTable could be assigned to the variable. If this is the case, a message box alerts the user and the procedure closes.
4. Determines whether there is more than one report filter field. (If the count of PageFields is greater than 1, there is more than one report filter.) This check is necessary for a simple reason: If you have multiple report filter fields, you might create hundreds of workbooks. The macro stops with a message box if the PivotTable contains more than one report filter field. You can remove this limitation simply by deleting or commenting out Step 4 in the macro.
5. Starts looping through each PageField and PivotItem. The outer loop tells Excel to iterate through all the report filters. The inner loop tells Excel to loop through all the pivot items in the report filter that currently is assigned.
6. For each pivot item, the report filter's CurrentPage property is changed to the currently assigned PivotItem's name. This alters the PivotTable report to match the pivot item.
7. Copies the PivotTable to the first worksheet in a new workbook. The TableRange1 property refers to the range of the main data area of the PivotTable. The new workbook is saved and closed. This example saves it to the path where the workbook containing the code is, but you can replace ThisWorkbook.Path with any directory you have access to.
8. Moves to the next pivot item. After the macro has looped through all pivot items in the report filter, the macro moves to the next PageField. After all PageFields have been assigned, the macro ends.

You can implement this kind of macro in a standard module. See the first section of this chapter, “Installing Macros,” for more information.

Resizing All Charts on a Worksheet

When building a dashboard, you often want to achieve some degree of symmetry and balance. This sometimes requires some level of chart size standardization. The macro in this section gives you an easy way to set a standard height and width for all your charts at once.

All charts belong to the ChartObjects collection. To take an action on all charts, you simply

iterate through all the charts in ChartObjects. Each chart in the ChartObjects collection has an index number that you can use to bring it into focus. For example, ChartObjects(1) points to the first chart in the sheet.

In this macro, you use this concept to loop through the charts on the active sheet with a simple counter. Each time a new chart is brought into focus, you change its height and width to the size you've defined.

```
Sub ResizeCharts()

'Step 1: Declare your variables.
  Dim i As Long

'Step 2: Start looping through all the charts.
  For i = 1 To ActiveSheet.ChartObjects.Count

'Step 3: Activate each chart and size.
    With ActiveSheet.ChartObjects(i)
      .Width = 300
      .Height = 200
    End With

'Step 4: Increment to move to next chart.
  Next i

End Sub
```

1. Declares the variable i as a Long integer used as the control variable in a For Next loop.
2. Starts the loop by setting i to count from 1 to the number of charts in the ChartObjects collection on the active sheet. When the code starts, i initiates with the number 1. As you loop, the variable increments up one number until it reaches a number equal to the number of charts on the sheet.
3. Refers to a ChartObject using i as the index number. The width and height of the chart is set to a number you specify in the code — 300 and 200 in this example. You can change these numbers to suit your needs.
4. Loops back to increment i up by one and get the next chart. After all charts have been processed, the macro ends.

To implement this macro, you can copy and paste it into a standard module. The first section in this chapter, "Installing Macros," outlines how to do this step.

Aligning a Chart to a Specific Range

Along with adjusting the size of charts, you can spend a good bit of time positioning them so that they align nicely in your dashboards. This macro helps easily align your charts to defined ranges, getting perfect positioning every time.

Every chart has four properties that dictate its size and position. These properties are Width, Height, Top, and Left. Interestingly enough, every Range object has these same properties. So if you set a chart's Width, Height, Top, and Left properties to match that of a particular range, the chart aligns to that range.

The idea is that after you have decided how you want your dashboard to be laid out, you take

note of the ranges that encompass each area of your dashboard. You then use those ranges in this macro to align each chart to the appropriate range. In this example, you adjust four charts so that their Width, Height, Top, and Left properties match a given range.

Note that you are identifying each chart with a name. Charts are, by default, named "Chart" and the order number they were added (Chart 1, Chart 2, Chart 3, and so on). You can see what each of your charts are named by clicking any chart, and then going choosing Format ⇒ Selection Pane. The task pane that opens (shown in Figure 8-5) lists all the charts on the sheet.

You can use it to get the appropriate chart names for your version of this macro.

```
Sub AlignCharts()

  Dim AlignRange As Range

  Set AlignRange = ActiveSheet.Range("B6:G19")
  With ActiveSheet.ChartObjects("Chart 1")
    .Height = AlignRange.Height
    .Width = AlignRange.Width
    .Top = AlignRange.Top
    .Left = AlignRange.Left
  End With

Set AlignRange = ActiveSheet.Range("B21:G34")
  With ActiveSheet.ChartObjects("Chart 2")
    .Height = AlignRange.Height
    .Width = AlignRange.Width
    .Top = AlignRange.Top
    .Left = AlignRange.Left
  End With

Set AlignRange = ActiveSheet.Range("I6:Q19")
  With ActiveSheet.ChartObjects("Chart 3")
    .Height = AlignRange.Height
    .Width = AlignRange.Width
    .Top = AlignRange.Top
    .Left = AlignRange.Left
  End With

Set AlignRange = ActiveSheet.Range("I21:Q34")
  With ActiveSheet.ChartObjects("Chart 4")
    .Height = AlignRange.Height
    .Width = AlignRange.Width
    .Top = AlignRange.Top
    .Left = AlignRange.Left
  End With

End Sub
```

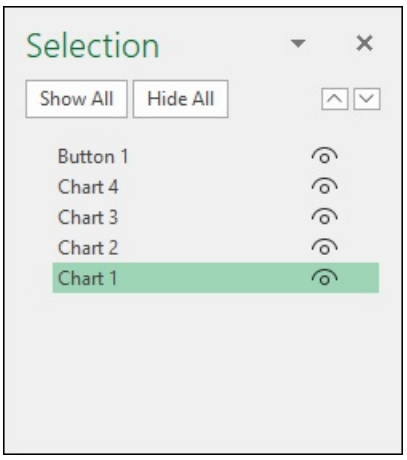


FIGURE 8-5: The Selection pane allows you to see all of your chart objects' names.

To implement this macro, you can copy and paste it into a standard module. See the "Installing Macros" section at the beginning of this chapter for more information.

Creating a Set of Disconnected Charts

When you need to copy charts from a workbook and paste them elsewhere (another workbook, PowerPoint, Outlook, and so on), it's often best to disconnect them from the original source data. This way, you won't get any of the annoying missing link messages that Excel throws. This macro copies all of the charts in the active sheet, pastes them into a new workbook, and disconnects them from the original source data.

This macro uses the ShapeRange.Group method to group all the charts on the active sheet into one shape. This is the VBA equiavalent to what you would do if you were to group a set of shapes manually. After the charts are grouped, you copy the group and paste it into a new workbook. You then use the BreakLink method to remove references to the original source data. When you do this, Excel hard-codes the chart data into array formulas.

```
Sub CreateDisconnectedCharts()

'Step 1: Declare your variables.
  Dim wbLinks As Variant

'Step 2: Group the charts, copy the group, and then ungroup.
  With ActiveSheet.ChartObjects.ShapeRange.Group
    .Copy
    .Ungroup
  End With

'Step 3: Paste into a new workbook and ungroup.
  Workbooks.Add.Sheets(1).Paste
  Selection.ShapeRange.Ungroup

'Step 4: Break the links.
  wbLinks = ActiveWorkbook.LinkSources(Type:=xlLinkTypeExcelLinks)
  ActiveWorkbook.BreakLink Name:=wbLinks(1), _
    Type:=xlLinkTypeExcelLinks

End Sub
```

1. Declares the wbLinks variant variable to hold the link source so it's easier to refer to it when breaking the links.
2. Uses ChartObjects.ShapeRange.Group to group all the charts into a single shape. Then the group is copied to the clipboard. After the group is copied, the charts are ungrouped. Grouping the charts before you copy preserves their relative position.
3. Pastes the copied group to the first worksheet in a new workbook. After the group has been pasted, they can be ungrouped so that each chart is separate again. Note that the newly created workbook is now the active workbook, so all references to ActiveWorkbook points back to this workbook.
4. Stores the link source in the wbLinks variable. The BreakLink method breaks the links, passing the first source.



REMEMBER Note that because this technique converts the chart source links to an array formula, this technique can fail if your chart contains too many data points. How many is too many? It can be different for every PC because it is limited by memory.

To implement this macro, you can copy and paste it into a standard module:

Printing All Charts on a Worksheet

To print a chart, you can click any embedded chart in your worksheet and then click Print. This prints the chart on its own sheet without any of the other data on the sheet. This sounds easy enough, but it can become a chore if you've got to do this for many charts. This macro makes short work of this task.

All charts belong to the ChartObjects collection. To take an action on all charts, you simply iterate through all the charts in ChartObjects. Each chart in the ChartObjects collection has an index number that you can use to refer to it. For example, ChartObjects(1) points to the first chart in the sheet.

In this macro, you use this concept to loop through the charts on the active sheet with a simple counter. Each time a new chart is referred to, print it.

```
Sub PrintAllCharts()

'Step 1: Declare your variable.
  Dim cht As ChartObject

'Step 2: Start looping through all the charts.
  For Each cht In ActiveSheet.ChartObjects

'Step 3: Print each chart.
    cht.Chart.PrintOut Copies:=1

'Step 4: Move to next chart.
  Next cht

End Sub
```

1. Declares a ChartObject variable named cht to be used as a control variable in the loop.
2. Starts looping through each ChartObject on the ActiveSheet. Each time the loop iterates, a different ChartObject is assigned to the cht variable.
3. Prints the chart using the PrintOut method.

   REMEMBER The ChartObject object doesn't have a PrintOut method, but it does have a Chart property that refers to the Chart object inside it. And that object has a PrintOut method.
4. Loops to get the next chart. After all charts have been printed, the macro ends.

To implement this macro, you can copy and paste it into a standard module, as shown in the

first section of this chapter, “Installing Macros.”

Chapter 9

Sending Emails from Excel

IN THIS CHAPTER

- **Mailing the active workbook as an attachment**
- **Mailing a specific range as an attachment**
- **Mailing a single sheet as an attachment**
- **Sending mail with a link to your workbook**
- **Mailing all email addresses in your contact list**
- **Saving all attachments to a folder**
- **Saving certain attachments to a folder**

Excel and Outlook work seemlessly together — as you expect from two Microsoft applications. If you've sent or received an Excel workbook through Outlook, you've integrated the two programs, albeit manually. This chapter shows you a few examples of how you can integrate Excel and Outlook in a more automated fashion.



REMEMBER The macros in this chapter automate Microsoft Outlook. For these macros to work, you need Microsoft Outlook installed on your system.

Installing Macros

You can install the macros in this chapter in three places depending on what kind of macro it is and how available you want the macro to be. Event macros are put into the module of the object the event is tied to. Non-event macros go into a standard module and can be in the workbook in which they are meant to run or in a workbook called the Personal Macro Workbook if you want to be able to run them on any workbook.

Event macros

In VBA, objects have events. VBA "listens" for these events to happen and runs special event macros when it detects and event. For example, the workbook object has an Open event. When VBA detects that the workbook has been opened, it looks for a procedure called Workbook_Open in the ThisWorkbook module. If it's there, it runs that macro.

Most of the event macros that you'll use, and most of the examples in the book, are workbook or worksheet events. Workbook event macros go in the ThisWorkbook module and worksheet event macros go in the module for the sheet you want the event code to run on.

To install an event macro, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **In the Project Explorer, find your project/workbook name and click the plus sign next to it to see all the objects.**
3. **Double-click ThisWorkbook.**
4. **Select the event in the right drop-down list.**
5. **Type or paste the code in the module.**

Personal Macro Workbook

There is a special workbook called the Personal Macro Workbook named personal.xlsb. If you record a macro, you can choose this workbook as the destination of the recorded macro. The workbook doesn't exist until you first record a macro into it. After that, it lives in a folder and Excel loads that workbook every time it starts.

The benefit of a workbook that opens every time Excel starts is that all the macros in that workbook are always available to you. If you have a macro that works on the active sheet or whatever range is currently selected, that's a good candidate for the Personal Macro Workbook.

To install a macro in the Personal Macro Workbook, follow these steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click personal.xlb in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**



REMEMBER If you don't see personal.xlsb in the Project Explorer, record a macro to the Personal Macro Workbook and Excel will create it for you.

Standard macros

All the other macro examples in this book are installed in a specific workbook in a standard module. If it's not an event macro and it's not a macro you want in the Personal Macro Workbook, then install it using the following steps:

1. **Press Alt+F11 to open the VBE.**
2. **Right-click the project/workbook name in the Project Explorer.**
3. **Choose Insert ⇒ Module.**
4. **Type or paste the code in the newly created module.**

Mailing the Active Workbook as an Attachment

The most fundamental Outlook task you can perform through automation is sending an email. In the sample code shown in this section, the active workbook is addressed to two email recipients as an attachment.



REMEMBER Some of you may notice that this macro doesn't use the SendMail command native to Excel. With the SendMail command, you can send simple emailsdirectly from Excel. However, the SendMail command is not as robust as Outlook automation. SendMail does not allow you to attach files, or use the CC and BCC fields in the email, which makes the technique that this macro uses a superior method.

Because this code runs from Excel, you need to set a reference to the Microsoft Outlook Object Library. You can set the reference by opening the Visual Basic Editor in Excel and choosing Tools ⇒ References. Scroll down until you find the entry Microsoft Outlook *XX* Object Library, where the *XX* is your version of Outlook. Select the check box next to the entry.

```
Sub EmailActiveWb()

'Step 1: Declare your variables.
  Dim OLApp As Outlook.Application
  Dim OLMail As Outlook.MailItem

'Step 2: Open Outlook and start a new mail item.
  Set OLApp = New Outlook.Application
  Set OLMail = OLApp.CreateItem(olMailItem)
  OLApp.Session.Logon

'Step 3: Build our mail item and send.
  With OLMail
    .To = "admin@kusleika.com; dick@kusleika.com"
    .CC = ""
    .BCC = ""
    .Subject = "This is the Subject line"
    .Body = "Sample File Attached"
    .Attachments.Add ActiveWorkbook.FullName
    .Display
  End With

'Step 4: Memory cleanup.
  Set OLMail = Nothing
  Set OLApp = Nothing

End Sub
```

1. Declares two object variables. OLApp is an object variable that exposes the Outlook Application object. OLMail is an object variable that holds an email, known as a MailItem.
2. Starts a new Outlook session and assigns it to OLApp. A new MailItem object using the CreateItem method passes in the olMailItem constant and assigns it to OLMail. Use your

default credentials to log into an Outlook session.



REMEMBER Keywords such as Outlook, CreateItem, and olMailItem are defined withing the Outlook Object Library, the one you set a reference to at the beginning of this example. Without setting that reference, VBA would not know what these terms mean or how to use them.



REMEMBER If Outlook is already running, logging in using the default credentials isn't necessary. VBA uses whatever session is already running. It's good to include line 2, however, in case you need it and it doesn't hurt to include it.

3. Sets the properties of the mail item. The To, CC, and BCC properties take a string that are valid email addresses. For multiple email addresses, separate them with a semicolon. The Subject and Body properties also take strings. The Add method of the Attachments object is uses to add files to the email. The full file path passes to this method; in this case by using the FullName property. Finally, the Display method shows the email.



TIP Replace .Display with .Send to automatically send the email without reviewing.

4. Releases the objects assigned to your variables. This practice is generally goodto follow. It reduces the chance of any problems caused by rogue objects that may remain open in memory. As you can see in the code, you simply set the variables to Nothing.

To implement this macro, copy and paste it into a standard module, as outlined in the "Installing Macros" section at the beginning of this chapter.

Mailing a Specific Range as an Attachment

You may not always want to send your entire workbook through email. This macro demonstrates how to send a specific range of data rather than the entire workbook.

Because this code is run from Excel, you need to set a reference to the Microsoft Outlook Object Library. You can set the reference by opening the Visual Basic Editor in Excel and choosing Tools ⇒ References. Scroll down until you find the entry Microsoft Outlook *XX* Object Library, where the *XX* is your version of Outlook. Select the check box next to the entry.

```
Sub EmailRange()

'Step 1: Declare your variables.
  Dim OLApp As Outlook.Application
  Dim OLMail As Outlook.MailItem

'Step 2: Copy range, paste to new workbook, and save it.
  Sheets("Revenue Table").Range("A1:E7").Copy
  Workbooks.Add
```

```
  Range("A1").PasteSpecial xlPasteValues
  Range("A1").PasteSpecial xlPasteFormats
  ActiveWorkbook.SaveAs ThisWorkbook.Path & "\TempRangeForEmail.xlsx"

'Step 3: Open Outlook and start a new mail item.
  Set OLApp = New Outlook.Application
  Set OLMail = OLApp.CreateItem(olMailItem)
  OLApp.Session.Logon

'Step 4: Build our mail item and send.
  With OLMail
    .To = "admin@kusleika.com; dick@kusleika.com"
    .CC = ""
    .BCC = ""
    .Subject = "This is the Subject line"
    .Body = "Sample File Attached"
    .Attachments.Add (ThisWorkbook.Path & "\TempRangeForEmail.xlsx")
    .Display
  End With

'Step 5: Delete the temporary Excel file.
  ActiveWorkbook.Close SaveChanges:=True
  Kill ThisWorkbook.Path & "\TempRangeForEmail.xlsx"
'Step 6: Memory cleanup.
  Set OLMail = Nothing
  Set OLApp = Nothing

End Sub
```

1. Declares two object variables. OLApp is an object variable that exposes the Outlook Application object. OLMail is an object variable that holds an email, known as a MailItem.
2. Copies a specified range and pastes the values and formats to a new Excel file. Then the new file is saved, along with a file path and filename.
3. Starts a new Outlook session and assigns it to OLApp. A new MailItem object uses the CreateItem method to pass in the olMailItem constant and assigns it to OLMail. The default credentials are used to log in to the current Outlook session.

   Keywords such as Outlook, CreateItem, and olMailItem are defined within the Outlook Object Library, the one you set a reference to at the beginning of the macro. Without setting that reference, VBA would not know what these terms mean or how to use them.

   If Outlook is already running, logging in using the default credentials isn't necessary. VBA use whatever session is already running. It's good to include line 3, however, in case you need it and it doesn't hurt to include it.
4. Sets the properties of the mail item. The To, CC, and BCC properties take a string that are valid email addresses. For multiple email addresses, separate them with a semicolon. The Subject and Body properties also take strings. The Add method of the Attachments object adds files to the email.The full file path of the new file is passes to this method. The Display method shows the email. You can replace .Display with .Send to automatically send the email without reviewing.
5. Deletes the temporary Excel file you created after the email is sent so you don’t leave temp files.
6. Releases the objects assigned to your variables. This practice is generally good to follow. It reduces the chance of any problems caused by rogue objects that may remain open in memory. As you can see in the code, you simply set the variables to Nothing.

To implement this macro, copy and paste it into a standard module, as outlined in the "Installing Macros" section at the beginning of this chapter.

Mailing a Single Sheet as an Attachment

This example demonstrates how to send a specific worksheet of data rather than the entire workbook.

Because this code is run from Excel, you need to set a reference to the Microsoft Outlook Object Library. You can set the reference by opening the Visual Basic Editor in Excel and choosing Tools ⇒ References. Scroll down until you find the entry Microsoft Outlook *XX* Object Library, where the *XX* is your version of Outlook. Place a check in the check box next to the entry.

```
Sub EmailSheet()

'Step 1: Declare your variables.
  Dim OLApp As Outlook.Application
  Dim OLMail As Outlook.MailItem

'Step 2: Copy the worksheet, paste to new workbook, and save it.
  Sheets("Revenue Table").Copy
  ActiveWorkbook.SaveAs ThisWorkbook.Path & "\TempRangeForEmail.xlsx"

'Step 3: Open Outlook and start a new mail item.
  Set OLApp = New Outlook.Application
  Set OLMail = OLApp.CreateItem(olMailItem)
  OLApp.Session.Logon

'Step 4: Build our mail item and send.
  With OLMail
    .To = "admin@kusleika.com; dick@kusleika.com"
    .CC = ""
    .BCC = ""
    .Subject = "This is the Subject line"
    .Body = "Sample File Attached"
    .Attachments.Add (ThisWorkbook.Path & "\TempRangeForEmail.xlsx")
    .Display
  End With

'Step 5: Delete the temporary Excel file.
  ActiveWorkbook.Close SaveChanges:=True
  Kill ThisWorkbook.Path & "\TempSheetForEmail.xlsx"

'Step 6: Memory cleanup.
  Set OLMail = Nothing
  Set OLApp = Nothing

End Sub
```

1. Declares two object variables. OLApp is an object variable that exposes the Outlook Application object. OLMail is an object variable that holds an email, known as a MailItem.
2. Copies the specified sheet and pastes the values and formats to a new Excel file. That file is then saved, along with a file path and filename.
3. Starts a new Outlook session and assigns it to OLApp. A new MailItem object uses the CreateItem method passing in the olMailItem constant and assigns it to OLMail. The default credentials are used to log in to the current Outlook session.

   Keywords sucsh as Outlook, CreateItem, and olMailItem are defined withing the Outlook

Object Library, the one you set a reference to at the beginning of this example. Without setting that reference, VBA would not know what these terms mean or how to use them.

If Outlook is already running, logging in using the default credentials isn't necessary. VBA uses whatever session is already running. It's good to include line 3, however, in case you need it and it doesn't hurt to include it.

4. Sets the properties of the mail item. The To, CC, and BCC properties take a string that are valid email addresses. For multiple email addresses, separate them with a semicolon. The Subject and Body properties also take strings. The Add method of the Attachments object adds files to the email. The full file path is passed to this method, in this case by using the FullName property. Finally, the Display method shows the email. You can replace .Display with .Send to automatically send the email without reviewing.
5. Deletes the temporary Excel file you created after the email is sent so that you don’t leave temp files.
6. Releases the objects assigned to your variables. This practice is generally good to follow. It reduces the chance of any problems caused by rogue objects that may remain open in memory. As this macro shows, you simply set the variables to Nothing.

To implement this macro, copy and paste it into a standard module, as outlined in the “Installing Macros” section at the beginning of this chapter.

Sending Mail with a Link to Your Workbook

If you're on the same network as the recipient, you may prefer to send a link to the file rather than an attachment. This macro does just that.



REMEMBER The recipients need at least read access to the network or location tied to the link.

Keep in mind that because this code is run from Excel, you need to set a reference to the Microsoft Outlook Object Library. You can set the reference by opening the Visual Basic Editor in Excel and choosing Tools ⇒ References. Scroll down until you find the entry Microsoft Outlook *XX* Object Library, where the *XX* is your version of Outlook. Select the check box next to the entry.

```
Sub EmailLink()

'Step 1: Declare your variables.
  Dim OLApp As Outlook.Application
  Dim OLMail As Outlook.MailItem

'Step 2: Open Outlook and start a new mail item.
  Set OLApp = New Outlook.Application
  Set OLMail = OLApp.CreateItem(olMailItem)
  OLApp.Session.Logon

'Step 3: Build our mail item and send.
  With OLMail
    .To = "admin@kusleika.com; dick@kusleika.com"
    .CC = ""
    .BCC = ""
```

```
    .Subject = "Monthly Report Email with Link"
    .HTMLBody = _
    "<p>Monthly report is ready. Click the link to get it.</p>" & _
    "<p><a href=" & Chr(34) & "Z:\Downloads\MonthlyReport.xlsx" & _
    Chr(34) & ">Download Now</a></p>"
    .Display
  End With

'Step 4: Memory cleanup.
  Set OLMail = Nothing
  Set OLApp = Nothing

End Sub
```

1. Declares two object variables. OLApp is an object variable that exposes the Outlook Application object. OLMail is an object variable that holds an email, known as a MailItem.
2. Starts a new Outlook session and assigns it to OLApp. A new MailItem object uses the CreateItem method to pass in the olMailItem constant and assigns it to OLMail. The default credentials are used to log into a current Outlook session.

   Keywords such as Outlook, CreateItem, and olMailItem are defined withing the Outlook Object Library, the one you set a reference to at the beginning of this example. Without setting that reference, VBA would not know what these terms mean or how to use them.

   If Outlook is already running, logging in using the default credentials isn't necessary, VBA used whatever session is already running. It's good to include line 2, however, in case you need it and it doesn't hurt to include it.
3. Sets the properties of the mail item. The To, CC, and BCC properties take a string that are valid email addresses. For multiple email addresses, separate them with a semicolon. The Subject and HTMLBody properties also take strings. MailItem objects have a Body property that takes plain text.

   TIP If you want anything beyond plain text, such as formatting, use the HTMLBody property and pass in valid hypertext markup language (HTML). HTML is beyond the scope of this book, but there are many examples on the Internet.

   The Display method showd the email. You can replace .Display with .Send to automatically send the email without reviewing.
4. Releases the objects assigned to your variables. This practice is generally good to follow. It reduces the chance of any problems caused by rogue objects that may remain open in memory. As the macro shows, you simply set the variables to Nothing.

To implement this macro, copy and paste it into a standard module, as outlined in the "Installing Macros" section at the beginning of this chapter.

Mailing All Email Addresses in Your Contact List

Ever need to send a mass mailing such as a newsletter or a memo? Instead of manually entering each of your contacts' email addresses, you can run the the macros in this section. It sends one email, automatically adding all the email addresses in your contact list to the email.

Because this code is run from Excel, you need to set a reference to the Microsoft Outlook Object Library. You can set the reference by opening the Visual Basic Editor in Excel and choosing Tools ⇒ References. Scroll down until you find the entry Microsoft Outlook *XX* Object Library, where the *XX* is your version of Outlook. Select the check box next to the entry.

```
Sub EmailAllContacts()

'Step 1: Declare your variables.
  Dim OLApp As Outlook.Application
  Dim OLMail As Outlook.MailItem
  Dim Cell As Range
  Dim Contacts As Range
  Dim sBcc As String

'Step 2: Define the range to loop through.
  Set Contacts = Sheets("Contacts").Range("H2:H21")

'Step 3: Open Outlook.
  Set OLApp = New Outlook.Application
  Set OLMail = OLApp.CreateItem(olMailItem)
  OLApp.Session.Logon

'Step 4: Add each address in the contact list.
  With OLMail
    For Each Cell In Contacts.Cells
      sBcc = sBcc & Cell.Value & ";"
    Next Cell
    .BCC = sBcc
    .Subject = "Sample Email"
    .Body = "Sample file is attached"
    .Attachments.Add ActiveWorkbook.FullName
    .Display  'Change to .Send to send without reviewing
  End With

'Step 5: Memory cleanup.
  Set OLMail = Nothing
  Set OLApp = Nothing

End Sub
```

1. Declares four object variables. OLApp is an object variable that exposes the Outlook Application object. OLMail is an object variable that holds an email, known as a MailItem. Cell is an control variable used in a For Each loop. Contacts is an Range variable that holds the range where the contacts are.
2. Assigns Contacts to the range of cells that contains your email addresses. This is the range of cells you loop through to add email addresses to your email.
3. Starts a new Outlook session and assigns it to OLApp. A new MailItem object uses the CreateItem method to pass in the olMailItem constant and assigns it to OLMail. The default credentials are used to log into the current Outlook session.

   Keywords such as Outlook, CreateItem, and olMailItem are defined withing the Outlook Object Library, the one you set a reference to at the beginning of this macro. Without setting that reference, VBA would not know what these terms mean or how to use them.

   If Outlook is already running, logging in using the default credentials isn't necessary. VBA uses whatever session is already running. It's good to include line 3, however, in

case you need it and it doesn't hurt to include it.

4. Sets the properties of the mail item. Loops through the cells in the Contacts range adding each to the BCC property along with a semicolon to separate them. The Subject and Body properties take strings. The Add method of the Attachments object adds files to the email. The full file path is passed to this method; in this case, by using the FullName property. The Display method shows the email. You can replace .Display with .Send to automatically send the email without reviewing.
5. Releases the objects assigned to your variables. This practice is generally good to follow. It reduces the chance of any problems caused by rogue objects that may remain open in memory. As the macros shows, you simply set the variables to Nothing.

To implement this macro, copy and paste it into a standard module, as outlined in the "Installing Macros" section at the beginning of this chapter.

Saving All Attachments to a Folder

You may often find that certain processes lend themselves to the exchange of data via email. For example, you may send a budget template for each branch manager to fill out and send back to you via email. Well, if there are 150 branch members, it could be a bit of a pain to download all those email attachments.

The macro in this section demonstrates one solution to this problem. It uses automation to search for all attachments in the inbox and saves them to a specified folder.

Because this code is run from Excel, you need to set a reference to the Microsoft Outlook Object Library. You can set the reference by opening the Visual Basic Editor in Excel and choosing Tool ⇒ References. Scroll down until you find the entry Microsoft Outlook *XX* Object Library, where the *XX* is your version of Outlook. Select the check box next to the entry.

```
Sub SaveAllAttachments()

'Step 1: Declare your variables.
  Dim ns As Outlook.Namespace
  Dim Inbox As Outlook.Folder
  Dim MItem As Outlook.MailItem
  Dim Atmt As Outlook.Attachment
  Dim FileName As String
  Dim sPath As String

'Step 2: Set a reference to your inbox.
  Set ns = GetNamespace("MAPI")
  Set Inbox = ns.GetDefaultFolder(olFolderInbox)

'Step 3: Check for messages in your inbox; exit if none.
  If Inbox.Items.Count = 0 Then
    MsgBox "No messages in folder."
    Exit Sub
  End If

'Step 4: Create directory to hold attachments.
  On Error Resume Next
    sPath = Environ("userprofile") & "\Documents\Attachments\"
    MkDir sPath
  On Error GoTo 0

'Step 5: Start to loop through each mail item.
```

```
  For Each MItem In Inbox.Items

'Step 6: Save each attachement then go to the next attachment.
    For Each Atmt In MItem.Attachments
      FileName = sPath & Atmt.FileName
      Atmt.SaveAsFile FileName
    Next Atmt
  Next MItem

'Step 7: Memory cleanup.
    Set ns = Nothing
    Set Inbox = Nothing

End Sub
```

1. Declares six variables. ns is an object that exposes the MAPI namespace. The target mail folder is assigned to Inbox. MItem is a control variable in the outer loop and is assigned a MailItem for each iteration. Atmt is a control variable in the inner loop and is assigned an Attachment for each iteration. FileName is a string variable that holds the name of the attachment. sPath is a string variable that holds the path to the folder where you save the attachments.
2. Sets the MyInbox variable to point to the inbox.
3. Makes sure actual messages are in the folder. If no messages are found, a message box states that there are no messages and the procedure closes.
4. Creates a directory to hold the attachments the macros finds. You can use an existing directory if you already have one. In that case, simply assign it to sPath. On Error Resume Next to ignores the error if the directory you are trying to create already exists.
5. Starts looping through each mail item in the target mail folder.
6. FLoops through each mail item for any attachments. Each attachment it finds gets saved into the specified directory you created.
7. Releases the objects assigned to your variables. This practice is generally good to follow. It reduces the chance of any problems caused by rogue objects that may remain open in memory. As the macro shows, you simply set the variables to Nothing.

To implement this macro, copy and paste it into a standard module.

Saving Certain Attachments to a Folder

The previous section showed how to use automation to search for all attachments in your inbox and save them to a specified folder. However, in most situations, you probably only want to save certain attachments; for example, those attached to emails that contain a certain subject. This macro demonstrates how to check for certain emails and save their attachments.

Because this code is run from Excel, you need to set a reference to the Microsoft Outlook Object Library. You can set the reference by opening the Visual Basic Editor in Excel and choosing Tools ⇒ References. Scroll down until you find the entry Microsoft Outlook *XX* Object Library, where the *XX* is your version of Outlook. Select the check box next to the entry.

```
Sub SaveAttachments()
```

```
'Step 1: Declare your variables.
  Dim ns As Outlook.Namespace
  Dim Inbox As Outlook.Folder
  Dim MItem As Outlook.MailItem
  Dim Atmt As Outlook.Attachment
  Dim FileName As String
  Dim sPath As String

'Step 2: Set a reference to your inbox.
  Set ns = GetNamespace("MAPI")
  Set Inbox = ns.GetDefaultFolder(olFolderInbox)

'Step 3: Check for messages in your inbox; exit if none.
  If Inbox.Items.Count = 0 Then
    MsgBox "No messages in folder."
    Exit Sub
  End If

'Step 4: Create directory to hold attachments.
  On Error Resume Next
    sPath = Environ("userprofile") & "\Documents\Attachments\"
    MkDir sPath
  On Error GoTo 0

'Step 5: Start to loop through each mail item.
  For Each MItem In Inbox.Items

'Step 6: Check for the words Data Submission in subject line.
    If InStr(1, MItem.Subject, "Data Submission") > 0 Then

'Step 7: Save each attachment; go to the next attachment.
      For Each Atmt In MItem.Attachments
        FileName = sPath & MItem.Subject & "-" & Atmt.FileName
        Atmt.SaveAsFile FileName
      Next Atmt
    End If
  Next MItem

'Step 8: Memory cleanup.
  Set ns = Nothing
  Set Inbox = Nothing

End Sub
```

1. Declares six variables. ns is an object used to expose the MAPI namespace. The target mail folder is assigned to Inbox. MItem is a control variable in the outer loop and is assigned a MailItem for each iteration. Atmt is a control variable in the inner loop and is assigned an Attachment for each iteration. FileName is a string variable that holds the name of the attachment. sPath is a string variable that holds the path to the folder where you save the attachments.
2. Sets the MyInbox variable to point to the inbox.
3. Makes sure actual messages are in the folder. If no messages are found, a message box states that there are no messages and the procedure closes.
4. Creates a directory to hold the attachments you find. You can use an existing directory if you already have one. In that case, simply assign it to sPath. On Error Resume Next ignores the error if the directory you are trying to create already exists.
5. Starts looping through each mail item in the target mail folder.
6. Uses the Instr function to check whether the string “Data Submission” is in the Subject line of the email. If that string does not exist, control is passed to the line following the End If.

7. Loops through and saves each attachment into the specified directory you created. The files are named with the subject, a dash, and the attachment name to help avoid files with the same name.
8. Releases the objects assigned to your variables. This practice is generally good to follow. It reduces the chance of any problems caused by rogue objects that may remain open in memory. As the macro shows, you simply set the variables to Nothing.

To implement this macro, copy and paste it into a standard module, as outlined in the "Installing Macros" section at the beginning of this chapter.

Chapter 10

Wrangling External Data with Macros

IN THIS CHAPTER

- **Working with external data connections**
- **Connecting to external databases**
- **Using macros to pull external data**
- **Working with external text files**

External data is exactly what it sounds like: data that isn't located in the Excel workbook in which you're working. Some examples of external data sources are text files, CSV files, Access tables, and even SQL Server tables.

There are numerous ways to get data into Excel. In fact, between the functionality found in the UI and the VBA/code techniques, there are too many techniques to focus on in one chapter. Instead, this chapter focuses on a handful of techniques that you can implement in most situations and that don't come with a lot of pitfalls and gotchas.

Working with External Data Connections

Every Excel workbook has the ability to store syntax that allows the workbook to pull in data from external databases, files, and websites. This ability is made possible by Excel's External Data Connection feature. You can create any number of data connections to retrieve the needed external data.

There are two main ways to get external data into Excel: PowerQuery and legacy wizards. PowerQuery is a relatively new method and is beyond the scope of this book. The legacy wizards are not on the Ribbon by default. To add the legacy wizards, choose File ⇒ Options ⇒ Data and select the legacy wizards you want to add to the Ribbon. Figure 10-1 shows the Excel Options dialog box with the From Text (Legacy) and From Access (Legacy) options checked.

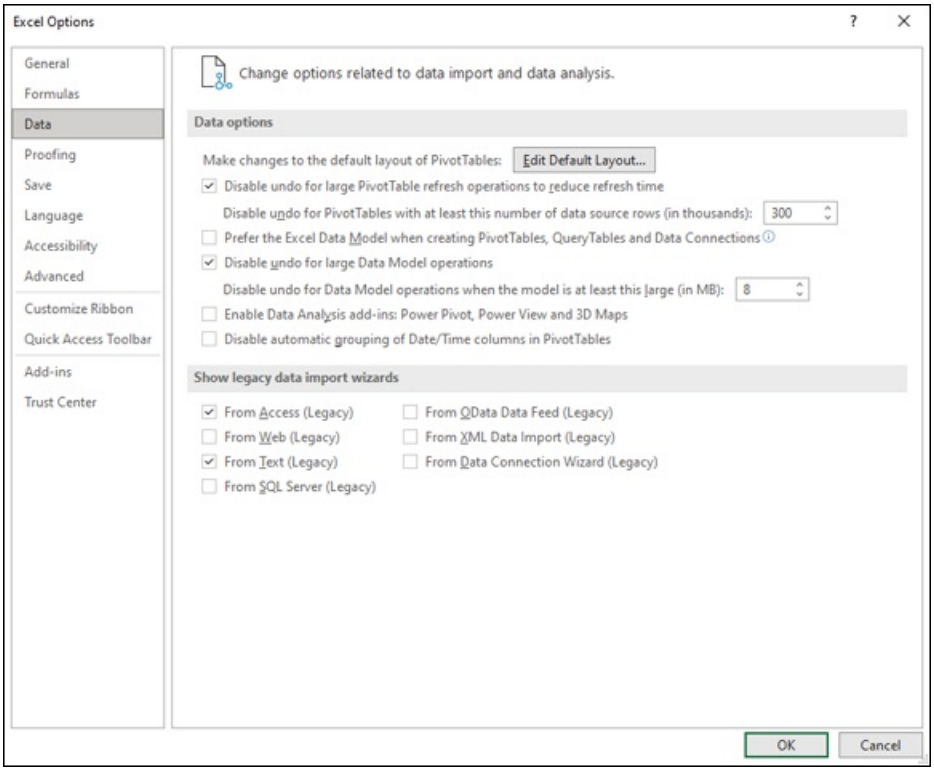


FIGURE 10-1: Show legacy wizards in the Ribbon.

This next section walks you through the ins and outs of working with external data connections.

Manually creating a connection

Excel has made it easy to manually connect to external data sources such as Microsoft Access, SQL Server, or any other ODBC connection you regularly use. For example, you can connect to an Access database by following these steps:



TIP Follow along by using the Facility Services.accdb Access database found on the book's website. The DynamicDataConnection.xlsm file contains the sample macros found in this section.

1. **Open a new Excel workbook and click the Data tab on the Ribbon.**
2. **In the Get & Transform Data group, choose Get Data ⇒ Legacy Wizards ⇒ From Access (Legacy).**

   The Select Data Source dialog box opens, as shown in Figure 10-2. If the database from which you want to import data is local, browse to the file's location and select it. If your target Access database resides on a network drive at another location, you need the proper authorization to select it.
3. **Navigate to your sample database and click Open.**

The Select Table dialog box shown in Figure 10-3 opens. This dialog box lists all the available tables and queries in the selected database.

The Select Table dialog box contains a column called Type. You can work with two types of Access objects: views and tables. View indicates that the dataset listed is an Access query, and Table indicates that the dataset is an Access table. In this example, Sales_By_Employee is actually an Access query. This means that you import the results of the query. This is true interaction at work; Access does all the back-end data management and aggregation, and Excel handles the analysis and presentation.

4. **Select your target table or query and click OK.**



REMEMBER In cases when your Access database is password protected, Step 3 starts a series of Data Link Properties dialog boxes asking for credentials (that is, username and password). Most Access databases don't require logon credentials, but if your database does require a username and password, type them in the Data Link Properties dialog box.

The Import Data dialog box shown in Figure 10-4 opens. Here you define where and how to import the table. You have the option of importing the data into a Table, a PivotTable Report, or a PivotChart. You also have the option of creating only the connection, making the connection available for later use.

Note that if you choose PivotChart or PivotTable Report, the data is saved to a pivot cache without writing the actual data to the worksheet. Thus your PivotTable can function as normal without you having to import potentially hundreds of thousands of data rows twice (once for the pivot cache and once for the spreadsheet).

5. **Select Table as the output view and define cell A1 as the output location, as shown in Figure 10-4.**
6. **Click OK.**

Your reward for all your work is a table (see Figure 10-5) that contains the imported data from your Access database.

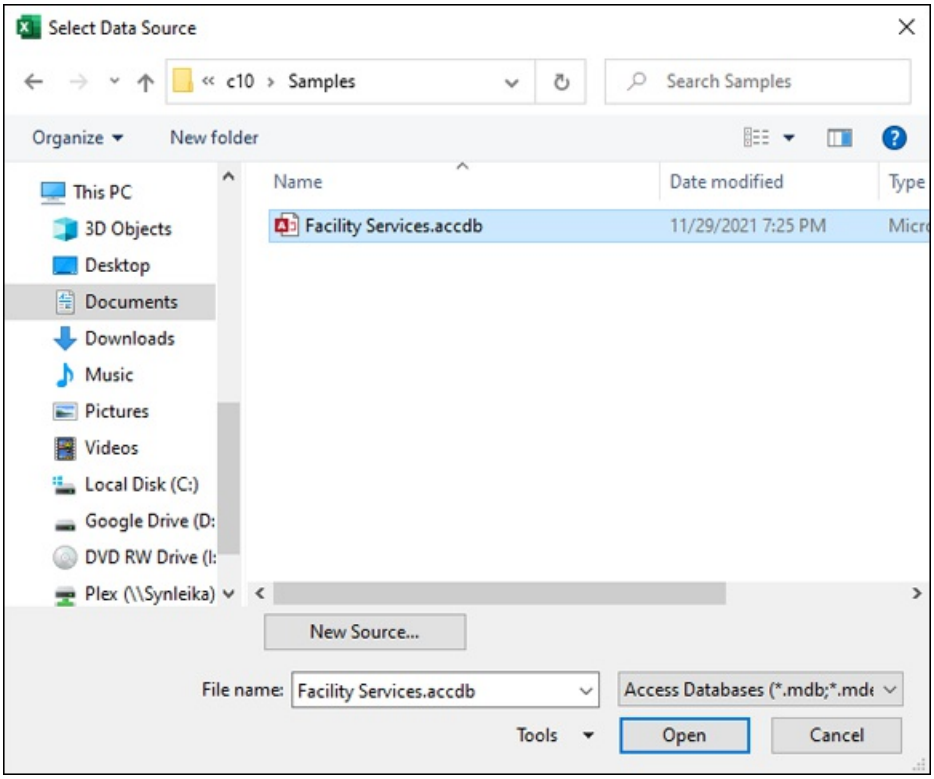


FIGURE 10-2: Choose the source database that contains the data you want imported.

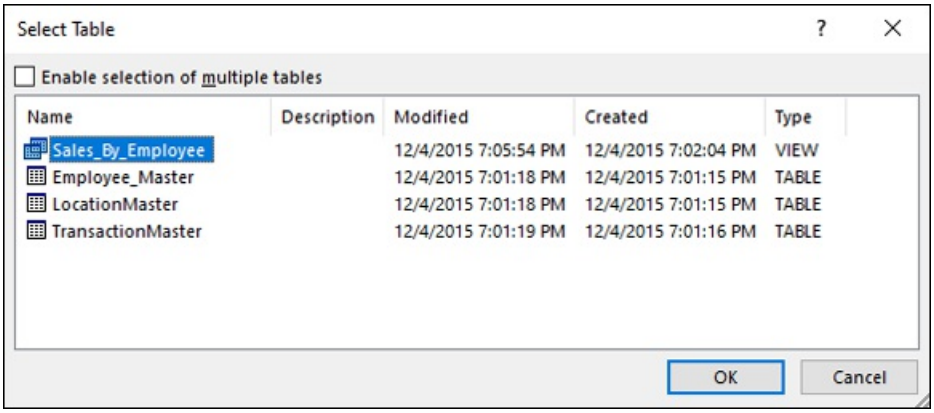


FIGURE 10-3: Select the Access object you want to import.

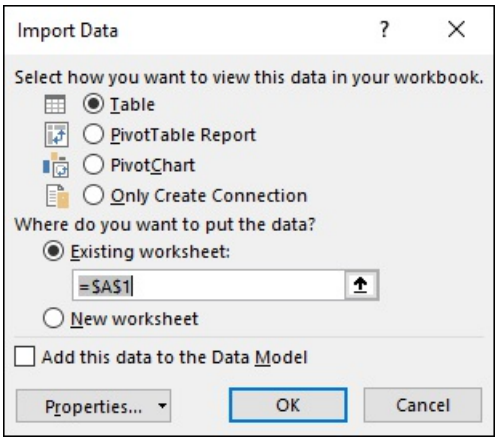


FIGURE 10-4: Choosing how and where to view your Access data.

	A	B	C	D
1	Region	Market	Branch_Number	Employee_Number
2	MIDWEST	TULSA	401612	1336
3	MIDWEST	TULSA	401612	1336
4	MIDWEST	TULSA	401612	60224
5	MIDWEST	TULSA	401612	60224
6	MIDWEST	TULSA	401612	55662
7	MIDWEST	TULSA	401612	60224
8	MIDWEST	TULSA	401612	1336
9	MIDWEST	TULSA	401612	55662
10	MIDWEST	TULSA	401612	55662
11	MIDWEST	TULSA	401612	1336
12	MIDWEST	TULSA	401612	55662
13	MIDWEST	TULSA	401612	55662

FIGURE 10-5: Data imported from Access.

The incredibly powerful thing about importing data this way is that it's refreshable. That's right. If you import data from Access using this technique, Excel creates a table that you can update by right-clicking it and selecting Refresh, as shown in Figure 10-6. When you update your imported data, Excel reconnects to your Access database and imports the data again. As long as a connection to your database is available, you can refresh with a mere click of the mouse.

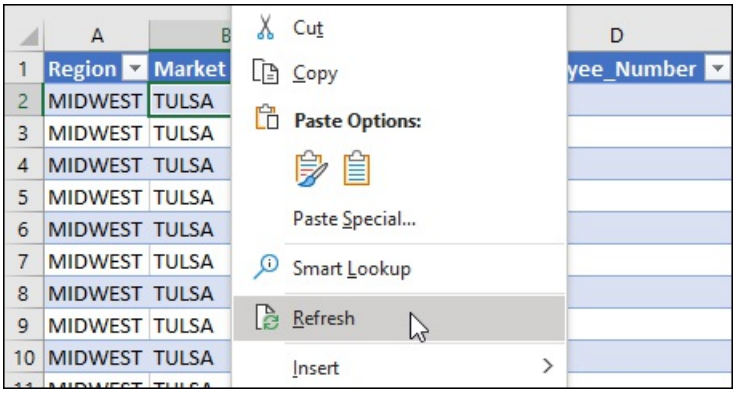


FIGURE 10-6: As long as a connection to your database is available, you can update your table with the latest data.



REMEMBER Again, a major advantage to using the Get & Transform Data group is that you can establish a refreshable data connection between Excel and Access. In most cases, you can set up the connection one time and then just update the data connection when needed. You can even record an Excel macro to update the data on some trigger or event, which is ideal for automating the transfer of data from Access.

Manually editing data connections

Once you have a connection, you can use the connection properties to point to another database table or query. You can even write your own SQL statements. *SQL* (Structured Query Language) is the language that relational database systems (such as Microsoft Access) use to perform various tasks. You can pass instructions right from Excel by using SQL statements. This can give you more control over the data you pull into your Excel model.

Although a detailed discussion of SQL is beyond the scope of this book, you can edit your external data connection using a simple SQL statement to pull in a different set of data.

1. **Right-click anywhere in the table and choose Table ⇒ Edit Query to show the Edit OLE DB Query dialog box as shown in Figure 10-7.**
2. **Change the Command Type property to SQL and then enter your SQL statement. In this case, you can enter:**

   ```
   SELECT * FROM [Sales_By_Employee]
   WHERE ([Market] = 'Tulsa')
   ```

   This statement tells Excel to pull in all records from the Sales_By_Employee table where the Market equals Tulsa.
3. **Click OK to confirm your changes and close the Edit OLE DB Query dialog box.**

   Excel immediately triggers a refresh of your external connection, bringing in your new data.

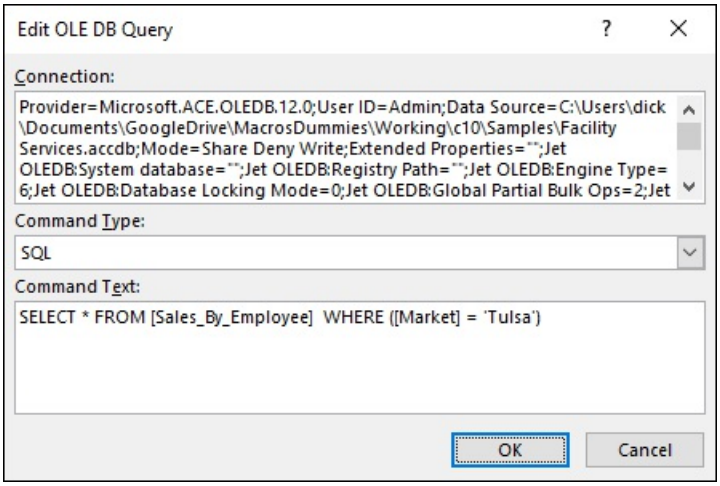


FIGURE 10-7: Select the SQL command type and enter your SQL statement.

Using Macros to Create Dynamic Connections

By now, you should have noticed that you haven't used any macros yet. So far, you have simply hard-coded the criteria for your connection. For example, in Figure 10-7, Tulsa is specified directly into the SQL statement WHERE clause. This obviously would cause the data being returned to always be data for Tulsa.

But what if you wanted to select a market and have the SQL statement dynamically change to respond to your selection? Well, you can use a bit of VBA to change the SQL statement on the fly. Follow these steps:

1. **Designate a cell in your worksheet that will catch the dynamic selection for your criteria.**

   For example, in Figure 10-8, cell C2 will be the place that users can select a market. You would typically give users a way to select criteria with either a Combo Box or a Data Validation list.

2. **Press Alt+F11 to open the VBE and choose Insert ⇒ Module from the menu bar**.

3. **Enter the following code in the newly created module:**

```
Sub RefreshQuery()

  Dim sSql As String
  Dim sh As Worksheet
  Dim lo As ListObject

  Set sh = ActiveWorkbook.Worksheets("Sheet1")
  Set lo = sh.ListObjects("Table_Facility_Servicess.accdb")

  sSql = "SELECT * FROM [Sales_By_Employee]"
  sSql = sSql & " WHERE ([Market] = '"
  sSql = sSql & sh.Range("C2").Value & "')"

  lo.QueryTable.CommandText = sSql

  lo.QueryTable.Refresh

End Sub
```

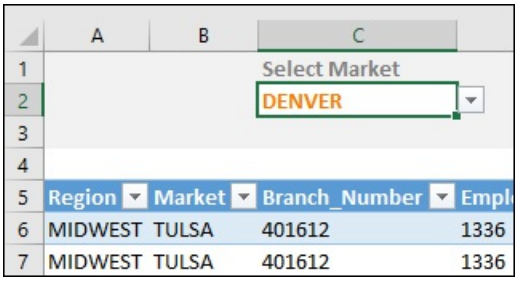
   A new macro called RefreshQuery is created. This macro uses one variable to hold a reference to the worksheet and another variable to hold a reference to the ListObject on that sheet. It changes the CommandText property of the QueryTable of the ListObject.

   A third variable holds the SQL statement that it assigns to the CommandText property. In this example, the command text selects from the [Sales_By_Employee] table and sets the criteria for the [Market] field to the value in cell C2. The code then refreshes the QueryTable.

4. **Close the VBE and place a new command button on your worksheet. To do this, click the Developer tab, select the Insert drop-down list, and add a Button Form control.**

5. **Assign the newly created RefreshQuery macro to the command button.**

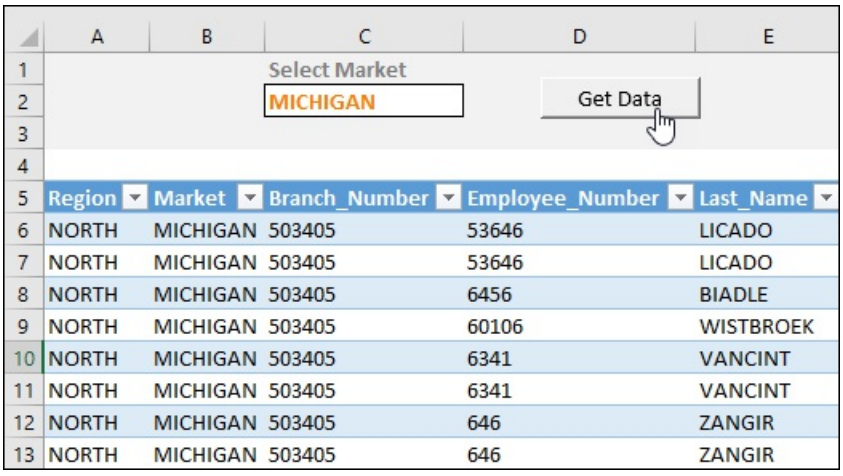
   See Chapter 1 if you need help on assigning the macro to a command button.



	A	B	C	
1			Select Market	
2			DENVER	
3				
4				
5	Region	Market	Branch_Number	Empl
6	MIDWEST	TULSA	401612	1336
7	MIDWEST	TULSA	401612	1336

FIGURE 10-8: Designate a cell that will trap the criteria selection.

If all went smoothly, you have a mechanism that allows for the dynamic extraction of data from your external database based on the criteria you specified (see Figure 10-9).



	A	B	C	D	E
1			Select Market		
2			MICHIGAN	Get Data	
3					
4					
5	Region	Market	Branch_Number	Employee_Number	Last_Name
6	NORTH	MICHIGAN	503405	53646	LICADO
7	NORTH	MICHIGAN	503405	53646	LICADO
8	NORTH	MICHIGAN	503405	6456	BIADLE
9	NORTH	MICHIGAN	503405	60106	WISTBROEK
10	NORTH	MICHIGAN	503405	6341	VANCINT
11	NORTH	MICHIGAN	503405	6341	VANCINT
12	NORTH	MICHIGAN	503405	646	ZANGIR
13	NORTH	MICHIGAN	503405	646	ZANGIR

FIGURE 10-9: You now have an easy-to-use mechanism to pull external data for a specified market.

Iterating through All Connections in a Workbook

You can use the Workbook.Connections collection to iterate through all the connection objects in a workbook and examine or modify their properties. For example, say you have a workbook with multiple external data connections. You can use this macro to populate a worksheet with a list of all connection objects in the current workbook, along with their associated connection strings and command texts:

```
Sub ListConnections()

'Step 1:  Declare your variables.
  Dim i As Long
  Dim Cn As WorkbookConnection

'Step 2:  Add a worksheet with header for
    'Connection Name, Connection String, and Command Text.
  Worksheets.Add
  With ActiveSheet.Range("A1:C1")
    .Value = Array("Cn Name","Connection String","Command Text")
    .EntireColumn.AutoFit
  End With

'Step 3: Iterate through each connection and write the
'needed properties to the newly created worksheet.
```

```
  For Each Cn In ThisWorkbook.Connections
   i = i + 1

   Select Case Cn.Type
    Case Is = xlConnectionTypeODBC

    With ActiveSheet
     .Range("A1").Offset(i,0).Value=Cn.Name
     .Range("A1").Offset(i,1).Value=Cn.ODBCConnection.Connection
     .Range("A1").Offset(i,2).Value=Cn.ODBCConnection.CommandText
    End With

    Case Is = xlConnectionTypeOLEDB

    With ActiveSheet
     .Range("A1").Offset(i,0).Value=Cn.Name
     .Range("A1").Offset(i,1).Value=Cn.OLEDBConnection.Connection
     .Range("A1").Offset(i,2).Value=Cn.OLEDBConnection.CommandText
    End With

   End Select

  Next Cn
End Sub
```

1. Declares two variables: a Long integer variable that ensures the data for each connection string is written on its own row, and a Workbook Connection object that exposes the properties you are looking for.
2. Adds a new worksheet along with column headers for each connection property you want to document.
3. Iterates through all the connections in the workbook, outputting the specified properties for each connection.



This step checks the kind of connection currently in focus. There are two types of connections: xlConnectionTypeODBC and xlConnectionTypeOLEDB. Because the syntax for getting their properties are slightly different, you need to test to check which type you are working with.



Check out Dynamic Data Connection.xlsm for a working version of this macro.

Using ADO and VBA to Pull External Data

Another technique for working with external data is to use VBA with ADO (ActiveX Data Objects). Using the combination of ADO with VBA allows you to work with external data sets in memory. This comes in handy when you need to perform complex, multilayered procedures and checks on external data sets, but you don’t want to create workbook connections or return those external data sets to the workbook.



REMEMBER When working with complex Excel workbooks that pull data from external sources, you will periodically encounter code (written by others) that uses ADO. It's important you recognize and understand the basics of ADO so you can deal with this kind of code. The next few sections walk you through some of the fundamental concepts of ADO and show you how to construct your own ADO procedures to pull data. Keep in mind that ADO programming is a broad topic that cannot be fully covered here. If you find that you need to work extensively with ADO and external data in your Excel application, you'll probably want to invest in one or more books that cover this topic in detail.

Understanding ADO syntax

When trying to grasp the basics of ADO, it helps to think of ADO as a tool that helps you accomplish two tasks: connect to a data source and specify the dataset to work with. The following section explores the fundamental syntax you need to know to do just that.

The connection string

The first thing you must do is connect to a data source. To do this, you must give VBA a few pieces of information. This information is passed to VBA in the form of a connection string. Here is an example connection string that points to an Access database:

```
"Provider=Microsoft.ACE.OLEDB.12.0;" & _
"Data Source= C:\MyDatabase.accdb;" & _
"User ID=Administrator;" & _
"Password=AdminPassword"
```

Don't be intimidated by all the syntax here. A connection string is fundamentally nothing more than a text string that holds a series of variables (also called arguments), which VBA uses to identify and open a connection to a data source. Although connection strings to either Access or Excel can get pretty fancy with a myriad of arguments and options, Ayou only need to focus on a handful of commonly used arguements: Provider, Data Source, Extended Properties, User ID, and Password.

- **Provider:** The Provider argument tells VBA what type of data source you are working with. When using Access or Excel as the data source, the Provider syntax reads: `Provider=Microsoft.ACE.OLEDB.12.0`.
- **Data Source:** The Data Source argument tells VBA where to find the database or workbook that contains the data needed. With the Data Source argument, you pass the full path of the database or workbook. For example: `Data Source=C:\Mydirectory\MyDatabaseName.accdb`.
- **Extended Properties:** The Extended Properties argument is typically used when connecting to an Excel workbook. This argument tells VBA that the data source is something other than a database. When working with an Excel workbook, this argument would read: `Extended Properties=Excel 12.0`.
- **User ID:** The User ID argument is optional and only used if a user ID is required to connect to the data source: `User Id=MyUserId`.

- **Password:** The Password argument is optional and only used if a password is required to connect to the data source: `Password=MyPassword`.

Take a moment now to re-examine the syntax previously shown. You can easily pick out the arguments in the connection string.

```
"Provider=Microsoft.ACE.OLEDB.12.0;" & _
"Data Source= C:\MyDatabase.accdb;" & _
"User ID=Administrator;" & _
"Password=AdminPassword"
```

Declaring a Recordset

In addition to building a connection to your data source, you need to define the data set you need to work with. In ADO, this dataset is referred to as the Recordset. A Recordset object is essentially a container for the records and fields returned from the data source. The most common way to define a Recordset is to open an existing table or query using the following arguments:

```
Recordset.Open Source, ConnectString, CursorType, LockType
```

- The **Source** argument specifies the data to be extracted. This is typically a table, query, or an SQL statement that retrieves records.
- The **ConnectString** argument specifies the connection string used to connect to your chosen data source.
- The **CursorType** argument defines how a Recordset allows you to move through the data to be extracted. In terms of pulling external data into Excel, the setting for this argument is adOpenForwardOnly. This CursorType is the most efficient type because it only allows you to move through the Recordset one way: from beginning to end. This is ideal for reporting processes where data only needs to be retrieved.
- The **LockType** argument lets you specify whether the data returned by the Recordset can be changed. This argument is typically set to adLockReadOnly (the default setting) to indicate that there is no need to edit the data returned. Alternatively, this argument can be set to adLockOptimistic, which allows for the free editing of the data returned (in the Recordset, not the source database).

The following shows the syntax for declaring a Recordset and opening the Products table:

```
MyRecordset.Open "Products", _
MyConnection, adOpenForwardOnly, adLockReadOnly
```

Using ADO in a macro

With these basic ADO fundamentals under your belt, you're ready to create your own ADO procedure. But before you do anything with ADO, you need to first set a reference to the ADO object library. Just as Excel has its own set of VBA objects, properties, and methods, so does ADO. Because Excel does not inherently know the ADO object model, you need to point Excel to the ADO reference library.

1. **Open a new Excel workbook and open the VBE.**

2. **Once you are in the VBE, choose Application ⇒ Tools ⇒ References.**

   The References dialog box, shown in Figure 10-10, opens.

3. **Scroll down until you locate the latest version of the Microsoft ActiveX Data Objects Library.**

   It's normal to have several versions of the same library displayed in the References dialog box, and generally best to select the latest version available.

4. **Click OK to confirm your selection.**

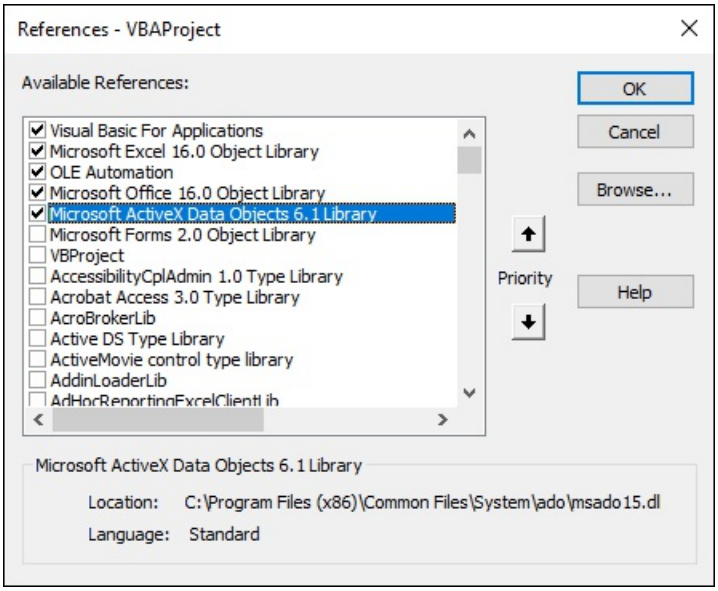


FIGURE 10-10: Select the latest version of the Microsoft ActiveX Data Objects Library.

You can open the References dialog box again to ensure that your reference is set. You knows that your selection took effect when the Microsoft ActiveX Data Objects Library appears near the top of the References dialog box with a check next to it.



REMEMBER The references you set in any given workbook or database are not applied at the application level. This means that you need to repeat these steps with each new workbook or database you create.

Now you can bring together everything in a macro. The following example uses ADO to connect to an Access database and retrieve the Products table:

```
Sub GetAccessData()

'Step 1: Declare your variables.
  Dim Cn As String
  Dim Rs As ADODB.Recordset

'Step 2: Define your connection string and open recordset.
  Cn = "Provider=Microsoft.ACE.OLEDB.12.0;" & _
    "Data Source=C:\MyDir\MyDatabaseName.accdb"

  Set Rs = New ADODB.Recordset
```

```
  Rs.Open "LocationMaster", _
  Cn, adOpenStatic, adLockReadOnly

'Step 3: Copy data from recordset to spreadsheet.
  Sheets("Sheet3").Range("A2").CopyFromRecordset Rs

'Step 4: Add data labels.
  With Sheets("Sheet3").Range("A1:C1")
    .Value = Array("Product", "Description", "Segment")
    .EntireColumn.AutoFit
  End With

End Sub
```

1. Declares two variables: a string variable to hold the connection string, and a Recordset object to hold the results of the data pull. In this example, the variable called Cn holds the connection string identifying the data source. Meanwhile, the variable called Rs holds the data returned by the procedure.
2. Defines the connection string for the ADO procedure. In this scenario, you are connecting to the MyDatabaseName.accdb file found in the C:\MyDir\ directory. Once you have defined your data source, you can open your Recordset and use MyConnect to return static read-only data.
3. Uses Excel's CopyFromRecordset method to get the data out of the RecordSet and into your spreadsheet. This method requires two pieces of information: the location of the data output and the Recordset object that holds the data. In this example, you are copying the data in the Rs object onto the sheet called Sheet3 (starting at cell A2).
4. Forces one final action where column headers are added by defining them in an array and writing them to the active sheet. (Otherwise, the CopyFromRecordset method doesn't return column headers or field names.)

With ADO and VBA, you can build all the necessary components one time in a nicely packaged macro, and then simply forget about it. As long as the defined variables in your code (that is, the data source path, the Recordset, the output path) do not change, then your ADO-based procedures require virtually zero maintenance.



TIP See Dynamic Data Connection.xlsm for a working version of this macro.

Working with Text Files

VBA contains a number of statements that allow for manipulation of files. These input/output (I/O) statements give you much more control over files than Excel's normal text file import and export options.

Opening a text file

The VBA Open statement opens a file for reading or writing. Before you can read from or write to a file, you must open it. When you use the Open statement, you can't see the file open like when you open it in Notepad. Instead, VBA loads the file into memory so it can access it.

The VBA Open statement is not the same as the Open method of the Workbook object. VBA's Open statement has its own set of arguments (shown as follows in bold). Here is the required syntax for the Open statement:

```
Open [pathname] For [mode] As [#]filenumber
```

- **pathname**: The pathname part of the Open statement simply contains the name and path (the path is optional) of the file to be opened.
- **mode**: The mode part of the Open statement specifies if and how the file can be edited or written to. This argument is a required field that can be any one of the following:
  - **Append:** This mode allows the file to be read and the data to be appended to the end of the file.
  - **Input:** This mode allows the file to be read but not written to.
  - **Output:** This mode always creates a new file that can be read from and written to.
  - **Binary:** A random-access mode that allows data to be read or written to on a byte-by-byte basis.
  - **Random:** A random-access mode that allows data to be read or written in units determined by an optional reclength argument of the Open statement.
- **filenumber**: This required argument is a file number ranging from 1 to 511. You can use the FreeFile function to get the next available file number.

The following example uses the Open statement to open a text file called MyFile as a read-only file designated as file #1:

```
Open "myfile.txt" For Input As #1
```

Reading the opened text file

After a text file has been opened, you typically want to either read the file or write to the file.

You can tell VBA to read a text file in three different ways. These statements are used for reading data from a sequential text file:

- **Input:** Reads a specified number of characters from a file.
- **Input #:** Reads data as a series of variables, with variables separated by a comma.
- **Line Input #:** Reads a complete line of data (delineated by a carriage return character, a linefeed character, or both).

Two statements are used for writing data to your opened text file:

- **Write #:** Writes a series of values, with each value separated by a comma and enclosed in quotes.
- **Print #:** Writes a series of values, with each value separated by a tab character.

A practical example: Logging workbook usage in a text file

If you're completely confused, don't worry. Things should clear up with a practical example. Say you want to log each time Excel is opened. You can create a macro that writes data to a text file every time Excel is opened and closed.

For this example to work reliably, the macro must be located in a workbook that's opened every time you start Excel. Storing the macro in your Personal Macro Workbook is an excellent choice.

The Personal Macro Workbook is loaded whenever you start Excel. In the VBE Project Explorer, it is named personal.xlsb.

1. **Press Alt+F11 to open the VBE.**
2. **In Project Explorer, find the personal.xlb project/workbook name and click the plus sign next to it and then click the plus sign next to the Microsoft Excel Objects folder to see all the objects.**
3. **Double-click ThisWorkbook.**
4. **Select the Open event in the Event drop-down list.**
5. **Type or paste the following code:**

```
Private Sub Workbook_Open()

  Open Application.DefaultFilePath & _
    "\excelusage.txt" For Append As #1

  Print #1, "Excel Started " & Now

  Close #1

End Sub
```



TIP See Excel Usage Log.xlsm for a working version of this macro.

This macro first opens a text file called excelusage.txt in your Windows Documents directory (Application.DefaultFilePath). If the text file does not exist, Excel creates it. After the text file is open, this macro appends a new line containing the current date and time and closes the file. The inserted line might look something like this:

```
Excel Started 4/5/2022 2:37:13 PM
```

A practical example: Importing a text file to a range

The example in this section opens and reads the TextFile.CSV. It then stores the values from the text file beginning at the active cell in the active worksheet. The code reads each character and essentially parses the line of data, ignoring quote characters and looking for commas to delineate the columns.

```
Sub ImportRange()

  Dim ImpRng As Range
  Dim Filename As String
  Dim r As Long, c As Long
  Dim txt As String, Char As String * 1
```

```
  Dim Data As String
  Dim i As Long
  Set ImpRng = ActiveCell
  On Error Resume Next
  Filename = Application.ThisWorkbook.Path & "\textfile.csv"

  Open Filename For Input As #1
  If Err <> 0 Then
    MsgBox "Not found: " & Filename, vbCritical, "ERROR"
    Exit Sub
  End If
  r = 0
  c = 0
  txt = ""

  Application.ScreenUpdating = False

  Do Until EOF(1)
    Line Input #1, Data
    For i = 1 To Len(Data)
      Char = Mid(Data, i, 1)
      If Char = "," Then 'comma
        ActiveCell.Offset(r, c) = txt
        c = c + 1
        txt = ""
      ElseIf i = Len(Data) Then 'end of line
        If Char <> Chr(34) Then txt = txt & Char
          ActiveCell.Offset(r, c) = txt
          txt = ""
      ElseIf Char <> Chr(34) Then
        txt = txt & Char
      End If
    Next i
    c = 0
    r = r + 1
  Loop

  Close #1
  Application.ScreenUpdating = True

End Sub
```



REMEMBER The preceding macro works with most data, but it has a flaw: It doesn't handle data that contains a comma or a quote character. But commas resulting from formatting are handled correctly (they’re ignored). In addition, an imported date is surrounded by number signs: for example, #2022-12-12#.

Part 5

The Part of Tens

IN THIS PART …

Look at a few tricks that help you more efficiently use the Visual Basic Editor.

Discover some debugging tips for avoiding errors in your VBA code.

Learn how to most effectively use the Excel Help system when searching for VBA help.

Gain insight into some of the resources available online to further your macro skills.

Chapter 11

Ten Handy Visual Basic Editor Tips

IN THIS CHAPTER

- **Applying block comments**
- **Copying multiple lines of code at once**
- **Jumping between modules and procedures**
- **Teleporting to your functions**
- **Staying in the right procedure**
- **Stepping through your code**
- **Stepping to a specific line in your code**
- **Stopping your code at a predefined point**
- **Seeing the beginning and end of variable values**
- **Turning off Auto Syntax Check**

If you're going to spend time working with macros in the Visual Basic Editor, then you should take advantage of a few of the built-in tools that make your job easier. Whether you're a fresh-faced analyst new to programming, or a jaded veteran living on Mountain Dew and sunflower seeds, these tips cangreatly improve your macro programming experience.

Applying Block Comments

Placing a single apostrophe in front of any line of code — commenting out code — effectively tells Excel to skip that line of code. Most programmers use the single apostrophe to create comments or notes within the code (see Figure 11-1).

```
'first char must be letter or underscore
Do Until LCase(Left$(sReturn, 1)) Like sLETTERS Or
    sReturn = Mid$(sReturn, 2, Len(sReturn))
Loop

'no spaces
sReturn = Replace(sReturn, Space(1), "")

'no slashes
sReturn = Replace(sReturn, "/", sUNDERSCORE)

'max 255 chars
sReturn = Left$(sReturn, lMAXLEN)

'not a cell ref
On Error Resume Next
    Set rTest = Range(sReturn)
On Error GoTo 0
```

FIGURE 11-1: A single apostrophe in front of any line turns that line into a comment.

It's sometimes beneficial to comment out multiple lines of code. This way, you can test certain lines of code while telling Excel to ignore the commented lines. For example, you might have code that manipulates some data and writes it to a text file. If you want to test the data manipulation part without creating the text file, you can comment out that part of the code.

Instead of spending time commenting out one line at a time, you can use the Edit toolbar to comment whole blocks of code.

You can display the Edit toolbar by going to the VBE menu and choosing View ⇒ Toolbars ⇒ Edit.

Select the lines of code you want commented out and then click the Comment Block icon on the Edit toolbar (see Figure 11-2).



TIP You can ensure the Edit toolbar is always visible by dragging it up to the VBE menu. It anchors itself to the location you choose. You can also double-click the toolbar's title bar to dock it to the menu area.

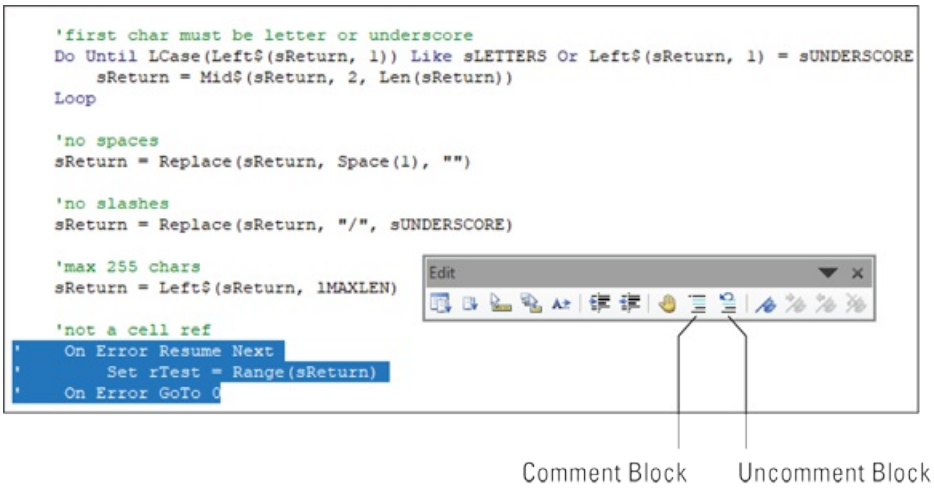


FIGURE 11-2: The Edit toolbar allows you to select entire blocks of code, and then apply comments to all the lines at once.

Copying Multiple Lines of Code at Once

You can copy entire blocks of code by highlighting the lines you need, and then holding down the Ctrl key while dragging the block where you need it. This is an old Windows trick that works even when you drag across modules.

You know you're dragging a copy when your cursor shows a plus symbol next to it, as shown in Figure 11-3.

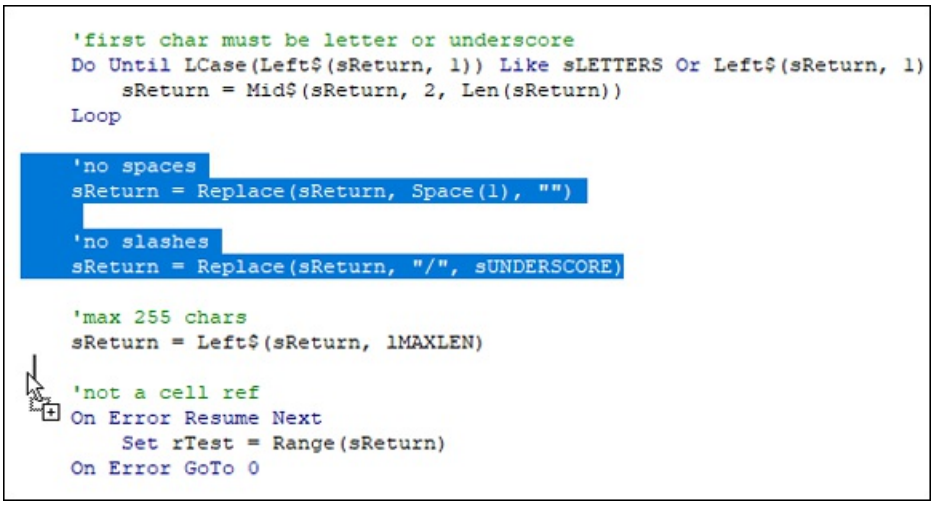


FIGURE 11-3: Holding down the Ctrl key while dragging code creates a copy of that code to where you point your cursor.

Jumping between Modules and Procedures

Once your cache of macro code starts to grow, it can be a pain to move quickly between modules and procedures. You can ease the pain by using a few hot keys:

- Press **Ctrl+Tab** to move quickly between modules.
- Press **Ctrl+Page Up** and **Ctrl+Page Down** to move between procedures within a module.
- Press **Ctrl+R** to select the current module in the Project Explorer and press the arrow keys to move to another module.

Teleporting to Your Functions

When reviewing a macro, you might sometimes encounter a variable or function name that is obviously pointing to some other piece of code. Instead of scouring through all the modules to find where that function or variable name comes from, you can simply place your cursor on that function/variable name, right-click, and choose Definition (or press Shift+F2).

As Figure 11-4 shows, you're instantly teleported to the origin of that function or variable name.

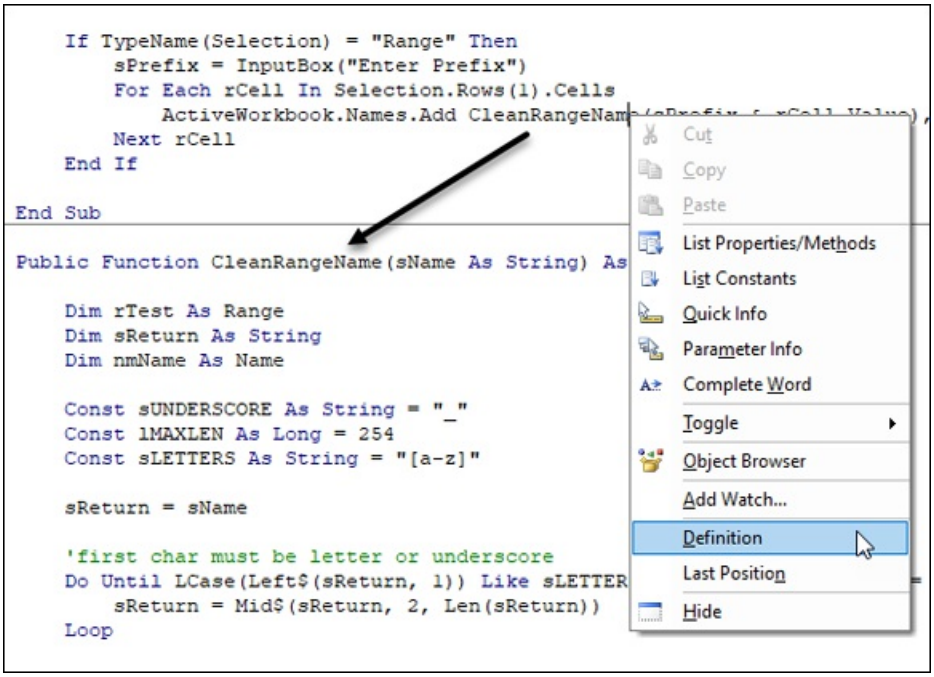


FIGURE 11-4: Pressing Shift+F2 with your cursor on a function or variable name will take you to it.

Right-click and choose Last Position (or press Ctrl+Shift+F2) to go back to where you started.

Staying in the Right Procedure

When your modules contain multiple procedures, you may find it less distracting to only work with one procedure at a time. The VBE provides two modes in which to work, the Procedure View mode and the Full Module View mode.

In Full Module View mode, all the procedures in the module are displayed in the Code pane and you scroll through the Code pane to find the procedure you want. In Procedure View mode, the Code pane displays only one procedure at a time (as shown in Figure 11-5). To switch procedures, use the drop-down lists at the top of the Code pane or press Ctrl+PgUp and Ctrl+PgDown. Use the two buttons at the bottom-left of the code pane to switch modes.

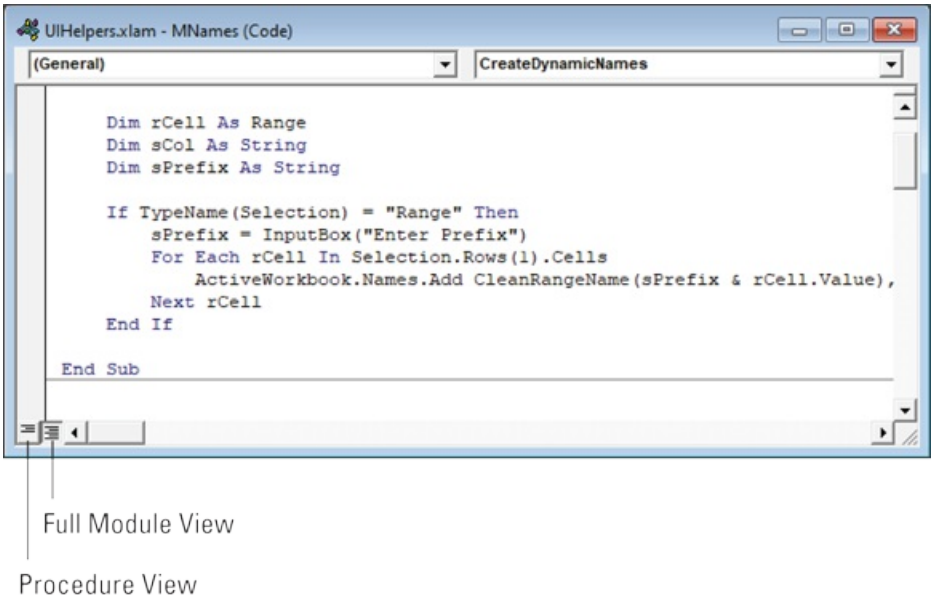


FIGURE 11-5: Use the Procedure View button to limit scrolling to just the active procedure.

Stepping through Your Code

VBA offers several tools to help you “debug” your code. In programming, the term debugging means finding and correcting possible errors in code.

One of the more useful debugging tools is the ability to step through your code one line at a time. When you step through code, you are literally watching as each line executes.

To step through your code, place your cursor anywhere within your macro and then press F8. The macro goes into debug mode.

The first line of code is highlighted and a small arrow appears on the Code pane's left margin (see Figure 11-6). Press F8 again to execute the highlighted line of code and move to the next line. You can keep pressing F8 to watch each line execute until the end of the macro.

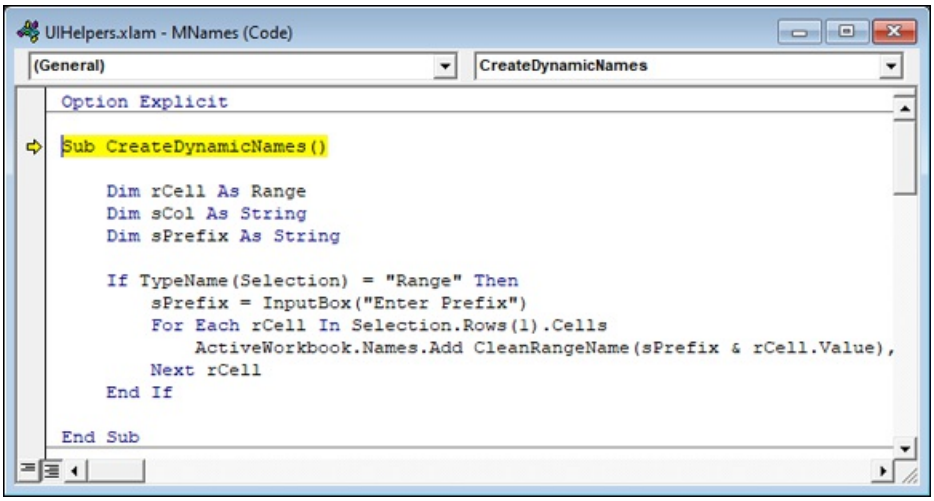


FIGURE 11-6: Press F8 key to step through each line of your macro at your own pace.



TIP While stepping through the code, you can hover over any string or numeric variable to see the current value of that variable.

To get out of debug mode, choose Debug ⇒ Step Out.

Stepping to a Specific Line in Your Code

In the last example, you saw how you can step through your code by placing your cursor anywhere within your macro and then pressing F8. This puts your macro into debug mode, highlighting the first line of code and placing a small arrow in the left margin of the Code pane.

This is great, but what if you want to start stepping through your code at a specific line? Well, you can do just that by simply moving the arrow!

When a line of code is highlighted in debug mode, you can click and drag the arrow in the left margin of the Code pane upward or downward, dropping it at whichever line of code you want to execute next (see Figure 11-7).

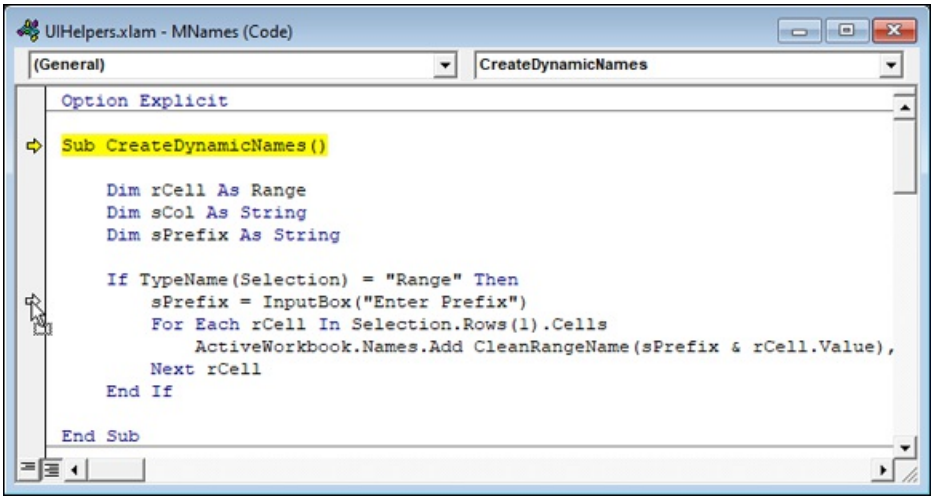


FIGURE 11-7: You can click and drag the yellow arrow while stepping through your code.



REMEMBER VBA executes your code from top to bottom, one line at a time. If you skip lines using this method, you might not get the results you expect.

Stopping Your Code at a Predefined Point

Another useful debugging tool is the ability to set a breakpoint in your code. When you set a breakpoint, your code runs as normal and then halts at the line of code you defined as the breakpoint.

This debugging technique comes in handy when you want to run tests on small blocks of code at a time. For example, if you suspect there may be an error in your macro but you know that the majority of the macro runs without any problems, you can set a breakpoint starting at the suspect line of code and start the macro. When the macros reaches your breakpoint, execution halts. At this point, you can press F8 to watch as the macro runs one line at a time.

To set a breakpoint in your code, place your cursor where you want the breakpoint to start, and then press F9. As Figure 11-8 shows, VBA clearly marks the breakpoint with a dot in the Code pane's left margin, and the code line itself is shaded maroon.

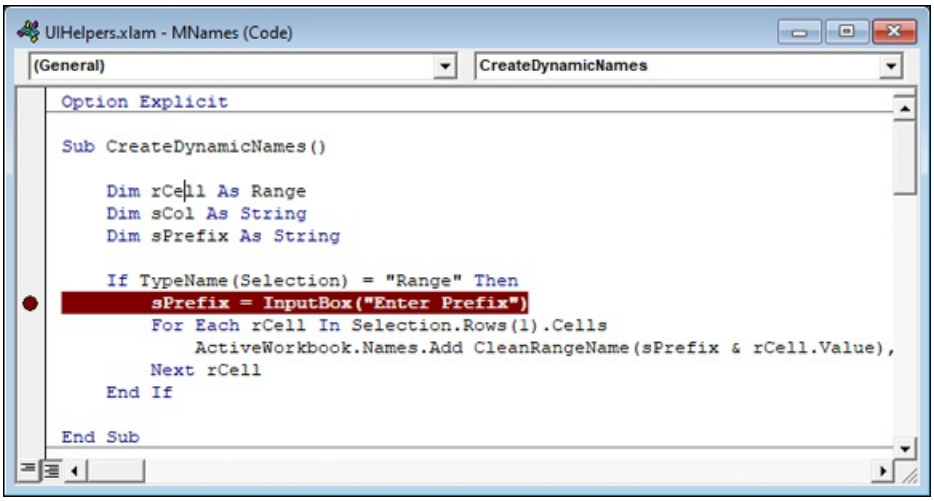


FIGURE 11-8: A breakpoint is marked by a dot in the left margin along with shaded text.



REMEMBER Hitting a breakpoint puts your macro into debug mode. To get out of debug mode, choose Debug ⇒ Step Out.

Seeing the Beginning and End of Variable Values

If you hover over a string or numeric variable in VBA while in debug mode, you can see the value of that variable in a tooltip. This allows you to see the values being passed in and out of variables — very useful.

However, these tooltips can hold only 77 characters (including the variable name). This means if the value in your variable is too long, it gets cut off.

To see the last 77 characters, hold down the Ctrl key on your keyboard while you hover.

Figure 11-9 shows what the tooltip looks like when hovering over a variable in debug mode.

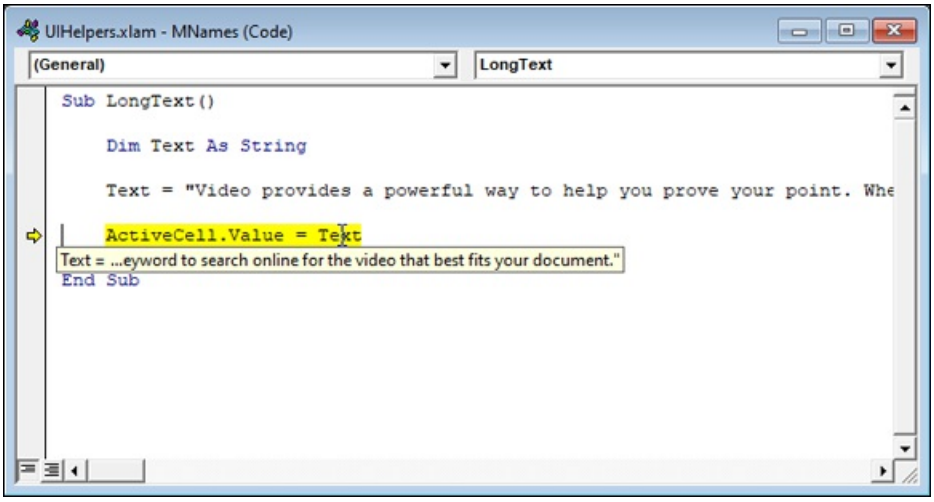


FIGURE 11-9: Showing the beginning and ending characters in a variable tooltip.

Turning Off Auto Syntax Check

Oftentimes, while working on some code, you find that you need to go to another line to copy something. You're not done with the line — you just need to leave it for a second. But the VBE immediately stops you in your tracks with an error message, similar to the one shown in Figure 11-10, warning you about something you already know.

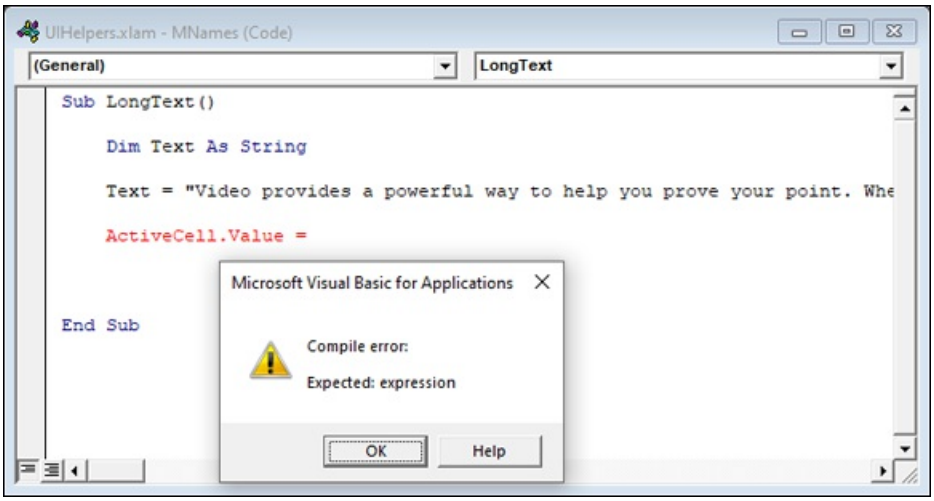


FIGURE 11-10: Leaving a line of code unfinished, even for a second, results in a jarring error message.

These message boxes force you to stop what you're doing to acknowledge the error by clicking OK. After a half-day of these abrupt message boxes, you'll be ready to throw your computer against the wall.

Well, you can save your computer and your sanity by turning off Auto Syntax Check. From the VBE menu, choose Tools ⇒ Options.

On the Editor tab of the Options dialog box, shown in Figure 11-11, uncheck the Auto Syntax Check option to stop these annoying error messages.

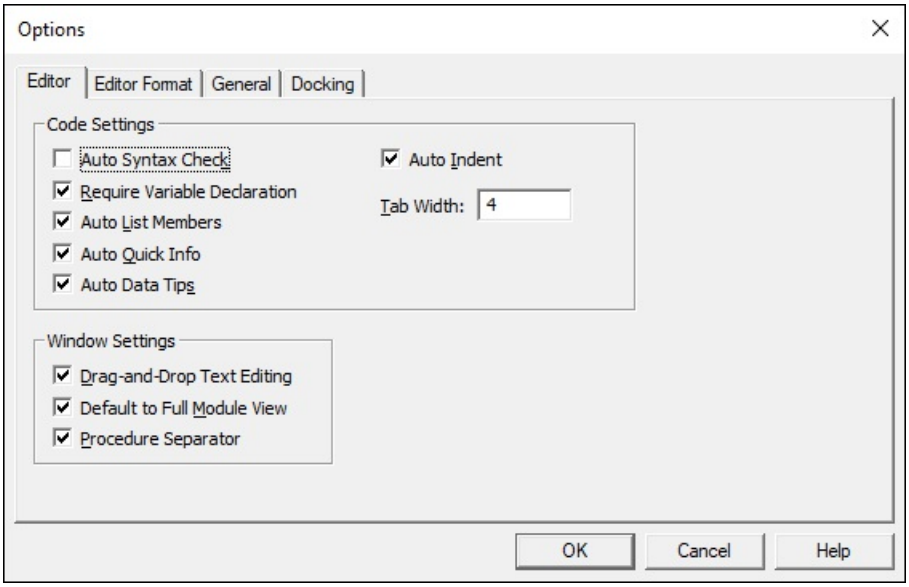


FIGURE 11-11: Uncheck the Auto Syntax Check option to prevent warning messages while coding.

Don’t worry about missing a legitimate mistake. Your code still turns red if you make a mistake, providing a visual indication that something is wrong.

Chapter 12

Ten Places to Turn for Macro Help

IN THIS CHAPTER

- **Letting Excel write the macro for you**
- **Using the VBA Help files**
- **Pilfering code from the Internet**
- **Leveraging user forums**
- **Visiting expert blogs**
- **Mining YouTube for video training**
- **Attending live and online training classes**
- **Learning from the Microsoft Office Dev Center**
- **Dissecting the other Excel files in your organization**
- **Asking your local Excel genius**

No one is going to be a macro expert in one day. VBA is a journey of time and practice. The good news is that plenty of resources are out there that can help you on your path. In this chapter, you discover ten of the most useful places to turn to when you need an extra push in the right direction.

Let Excel Write the Macro for You

One of the best places to get macro help is the Macro Recorder in Excel. When you record a macro with the Macro Recorder, Excel writes the underlying VBA for you. After recording, you can review the code; see what the recorder is doing, and then try to turn the code it creates into something more suited to your needs.

For example, say you need a macro that refreshes all the PivotTables in your workbook and clears all the filters in each PivotTable. Writing this macro from a blank canvas would be a daunting task. Instead, you can start the Macro Recorder and record yourself refreshing all the PivotTables and clearing all the filters. Once you've stopped recording, you can review the macro and make any changes you deem necessary.

Use the VBA Help Files

The Help system for Microsoft Office (Excel included) has improved immensely over the last few versions. And because it is either online or pulls from an online source, the Help system is generally up to date.

You need to remember two basic tenets of the Excel Help system:

- **Location matters when asking for help.** If you press F1 in Excel to show the Help task pane and type some VBA search terms, you won't get any useful results. When searching for VBA help, start from the VBE.
- **Online help is better than offline help.** Pressing F1 in the VBE opens a browser window (assuming you're connected to the Internet) and takes you to the VBA language reference website hosted by Microsoft. If your cursor happens to be on a keyword, object name, or property or method name, it might take you right to the relevant page.

Pilfer Code from the Internet

The dirty secret about programming in the Internet age is that there is no original code out there anymore. All the macro syntax that will ever be needed has been documented somewhere on the Internet. In many ways, programming has become less about the code one creates from scratch, and more about how to take existing code and apply it creatively to a particular scenario.

If you are stuck trying to create a macro for a particular task, open your favorite search engine and simply describe the task you're trying to accomplish. For the best results, enter "Excel VBA" before your description.

For example, if you are trying to write a macro that deletes all blank rows in a worksheet, search for "Excel VBA delete blank rows in a worksheet." You can bet two months' salary that someone out there on the Internet has already tackled the same problem. Nine times out of ten, you will find some example code that gives you the nugget of information you need to build your own macro.

Leverage User Forums

If you find yourself in a bind, you can post your question in a forum to get customized guidance based on your scenario.

User forums are online communities that revolve around a particular topic. In these forums, you can post questions and have experts offer advice on how to solve particular problems. The folks answering the questions are typically volunteers who have a passion for helping the community solve real-world challenges.

Many forums are dedicated to all things Excel. To find an Excel forum, enter the words "Excel Forum" in your favorite search engine.



TIP Here are a few tips for getting the most out of user forums:

- **Always read and follow the forum rules before you get started.** These rules often include advice on posting questions and community etiquette guidelines.

- **Check whether your question has already been asked and answered.** Save time by searching the forum's archive of questions to see whether your question has been asked before.
- **Use concise and accurate subject titles for your questions.** Don't create forum questions with vague titles such as "Need advice," or "Please Help."
- **Keep the scope of your questions as narrow as possible.** Don't ask questions such as "How do I build an invoicing macro in Excel?"
- **Be patient.** Remember the folks answering your questions are volunteers who typically have day jobs. Give the community some time to answer your question.
- **Check back often.** After posting your question, you may receive requests for more details about your scenario. Do everyone a favor and return to your posting to either review the answers or respond to follow-up questions.
- **Thank the expert who answered your question.** If you receive an answer that helps you, take a moment to post a thank you to the expert who helped you out.

Visit Expert Blogs

A few dedicated Excel gurus share their knowledge through blogs. These blogs are often treasure troves of tips and tricks, offering nuggets that can help build your skills. Best of all, they are free!

Although these blogs don't necessarily speak to your particular needs, they offer articles that advance your knowledge of Excel and can even provide general guidance on how to apply Excel in practical business situations.



TIP Here is a list of a few of the best Excel blogs on the Internet today:

- **ExcelGuru:** Ken Puls is a Microsoft Excel MVP who shares knowledge at his blog, www.excelguru.ca/blog. In addition to his blog, Ken offers several learning resources for advancing your knowledge in Excel.
- **Chandoo.org:** Purna "Chandoo" Duggirala is a Microsoft Excel MVP out of India, who burst onto the scene in 2007. His innovative blog (http://chandoo.org/) offers many free templates and articles aimed at "making you awesome in Excel."
- **Contextures:** Debra Dalgleish is a Microsoft Excel MVP and the owner of one of the most popular Excel sites, www.contextures.com. With an alphabetized list of over 350 Excel topics, you're sure to find something that will interest you.
- **DailyDose:** Yours truly is the owner of the longest running Excel blog, www.dailydoseofexcel.com. While I don't actively post anymore, there is over ten years' worth of articles and examples.
- **MrExcel:** Bill Jelen is the larger-than-life ambassador of Excel. This long-time Excel

MVP offers over a thousand free videos and a huge library of training resources on his site at www.mrexcel.com.

Mine YouTube for Video Training

Some of us learn better if we watch a task being done. If you find that you absorb video training better than online articles, consider mining YouTube. Dozens of channels ar run by amazing folks who have a passion for sharing knowledge. You'll be surprised at how many free high-quality video tutorials you can find.

Go to www.YouTube.com and search for the words "Excel VBA."

Attend Live and Online Training Classes

Live and online training events are an awesome way to absorb Excel knowledge from a diverse group of people. Not only is the instructor feeding you techniques, but the lively discussions during the class offer a wealth of ideas and new tips that you might never have thought of. If you thrive on the energy of live training events, then consider searching for Excel classes.



TIP Here are a few sites that provide excellent instructor-led Excel courses:

http://academy.excelhero.com/excel-hero-academy-tuition

http://chandoo.org/wp/vba-classes

https://exceljet.net

Learn from the Microsoft Office Dev Center

The Microsoft Office Dev Center is a site dedicated to helping new developers get a quick start in programming Office products. You can get to the Excel portion of this site by going to https://docs.microsoft.com/en-us/office/client-developer/excel/excel-home.

Dissect the Other Excel Files in your Organization

Like finding gold in your backyard, the existing files in your organization are often a treasure trove for learning. Consider cracking open those Excel files that contain macros, and have a look under the covers. See how others in your organization use macros. Try going through the macros line-by-line and see whether you can spot new techniques.

You could find a few new tricks you never thought of. You might even stumble upon entire

chunks of useful code you can copy and implement in your own workbooks.

Ask Your Local Excel Genius

Do you have an Excel genius in your company, department, organization, or community? Make friends with that person today. What you have there is your own personal Excel forum.

Most Excel experts love sharing their knowledge, so don’t be afraid to approach your local Excel guru to ask questions or seek out advice on how to tackle certain macro problems.

Chapter 13

Ten Ways to Speed Up Your Macros

IN THIS CHAPTER

- **Disabling performance draining Excel features**
- **Avoiding unnecessary macro actions**
- **Optimizing your macro code for better performance**

As your macros become increasingly robust and complex, you may find they lose performance. When discussing macros, the word performance is usually synonymous with speed. Speed is how quickly your VBA procedures perform their intended tasks.

You can take steps to improve the performance of your macros. In this chapter, you find ten ways you can help keep your Excel macros running at their optimum performance level.

Halt Calculations

Each time a cell that affects any formula in your spreadsheet is changed or manipulated, Excel recalculates the entire worksheet. In worksheets that have a large amount of formulas, this behavior can drastically slow down your macros.

If your workbook is formula intensive, you may not want Excel to trigger a recalculation every time your macro alters a cell value. You can use the Application.Calculation property to tell Excel to switch to manual calculation mode. When a workbook is in manual calculation mode, the workbook does not recalculate until you explicitly trigger a calculation by pressing the F9 key on your keyboard.

Turning off the automatic calculation behavior of Excel can dramatically speed up your macro. The idea is to place Excel into manual calculation mode, run your code, and then switch back to automatic calculation mode.

```
Sub Macro1()

  Application.Calculation = xlCalculationManual

'Place your macro code here

  Application.Calculation = xlCalculationAutomatic

End Sub
```



TIP Setting the calculation mode back to xlCalculationAutomatic automatically triggers a recalculation of the worksheet. So there is no need to press F9 after your macro runs.



WARNING Depending on the task at hand, you may actually need calculations to be performed during the running of your macro, in which case you would not want to invoke manual calculation mode. Be sure to think about your specific scenario and determine what will happen when calculations are turned off while your macro runs.

Disable Screen Updating

You may notice that when your macros run, your screen does a fair amount of flickering. This flickering is Excel trying to redraw the screen to show the current state the worksheet is in. Unfortunately, each time Excel redraws the screen, it takes time. In most cases, you don't need Excel using up resources to redraw the screen each time your macro performs some action.

In addition to setting the calculation mode to manual, you can use the Application.ScreenUpdating property to disable any screen updates until your macro has completed. This saves time and resources, allowing your macro to run a little faster. Once your macro code is done running, you can turn screen updating back on.

```
Sub Macro1()

  Application.Calculation = xlCalculationManual
  Application.ScreenUpdating = False

'Place your macro code here

  Application.Calculation = xlCalculationAutomatic
  Application.ScreenUpdating = True

End Sub
```



TIP After you set the ScreenUpdating property back to True, Excel not only resumes redrawing, but triggers the first redraw to show all the changes from the macro.

Turn Off Status Bar Updates

At the bottom of the Excel window, you see the Excel status bar. The status bar normally displays the progress of certain actions in Excel. For example, if you copy/paste a range, Excel shows the progress of that operation in the status bar. Oftentimes, the action is performed so fast, you don't see the status bar progress. However, if your macro is working with lots of data, the status bar takes up some resources.

It's important to note that turning off screen updating is separate from turning off the status bar display. That is to say, the status bar continues to be updated even if you disable screen updating. You can use the Application.DisplayStatusBar property to temporarily disable any status bar updates, further improving the performance of your macro.

```
Sub Macro1()
```

```
  Application.Calculation = xlCalculationManual
  Application.ScreenUpdating = False
  Application.DisplayStatusBar = False

'Place your macro code here

  Application.Calculation = xlCalculationAutomatic
  Application.ScreenUpdating = True
  Application.DisplayStatusBar = True

End Sub
```

Tell Excel to Ignore Events

As Chapter 3 discusses, you can implement macros as event procedures, telling Excel to run certain code when a worksheet or workbook changes.

Sometimes, standard macros make changes that will actually trigger an event procedure. For example, say you have a Worksheet_Change event implemented for Sheet1 of your workbook. Any time a cell or range is altered, the Worksheet_Change event fires.

So if you have a standard macro that manipulates several cells on Sheet1, each time a cell on that sheet is changed, your macro has to pause while the Worksheet_Change event runs. You can imagine how this behavior would slow down your macro.

You can add another level of performance-boosting by using the EnableEvents property to tell Excel to ignore events while your macro runs.

Simply set the EnableEvents property to False before running your macro. After your macro code is done running, you can set the EnableEvents property back to True.

```
Sub Macro1()

  Application.Calculation = xlCalculationManual
  Application.ScreenUpdating = False
  Application.DisplayStatusBar = False
  Application.EnableEvents = False

'Place your macro code here

  Application.Calculation = xlCalculationAutomatic
  Application.ScreenUpdating = True
  Application.DisplayStatusBar = True
  Application.EnableEvents = True

End Sub
```



WARNING Although disabling events can indeed speed up your macros, you may actually need some events to trigger while your macro runs. Be sure to think about your specific scenario and determine what will happen if your worksheet or workbook events are turned off while your macro runs.

Hide Page Breaks

Another opportunity for a performance boost can be found in page breaks. Each time your

macro modifies the number of rows, modifies the number of columns, or alters the page setup of a worksheet, Excel is forced to take up time recalculating where the page breaks are shown on the sheet.

You can avoid this by simply hiding the page breaks before starting your macro.

Set the DisplayPageBreaks sheet property to False to hide page breaks. If you want to continue to show page breaks after your macro runs, you can set the DisplayPageBreaks sheet property back to True.

```
Sub Macro1()

  Application.Calculation = xlCalculationManual
  Application.ScreenUpdating = False
  Application.DisplayStatusBar = False
  Application.EnableEvents = False
  ActiveSheet.DisplayPageBreaks = False

'Place your macro code here

  Application.Calculation = xlCalculationAutomatic
  Application.ScreenUpdating = True
  Application.DisplayStatusBar = True
  Application.EnableEvents = True
  ActiveSheet.DisplayPageBreaks = True

End Sub
```

Suspend PivotTable Updates

If your macro manipulates PivotTables that contain large data sources, you may experience poor performance when doing things such as dynamically adding or moving pivot fields. This is because each change you make to the structure of the PivotTable requires Excel to recalculate the values in the PivotTable for each pivot field your macro touches.

You can improve the performance of your macro by suspending the recalculation of the PivotTable until all your pivot field changes have been made. Simply set the PivotTable.ManualUpdate property to True to defer recalculation, run your macro code, and then set the PivotTable.ManualUpdate property back to False to trigger the recalculation.

```
Sub Macro1()

  ActiveSheet.PivotTables("PivotTable1").ManualUpdate=True

'Place your macro code here

  ActiveSheet.PivotTables("PivotTable1").ManualUpdate=False

End Sub
```

Steer Clear of Copy and Paste

It’s important to remember that while the Macro Recorder saves time by writing VBA code for you, it does not always write the most efficient code. A prime example of this is how the Macro Recorder captures any copy-and-paste action you perform while recording.

If you were to copy cell A1 and paste it into cell B1 while recording a macro, the Macro Recorder would capture this:

```
Range("A1").Select

Selection.Copy

Range("B1").Select

ActiveSheet.Paste
```

While this code indeed copies from cell A1 and pastes into B1, it forces Excel to use the clipboard, which adds a kind of middle man where there does not need to be one.

You can give your macros a slight boost by cutting out the middle man and performing a direct copy from one cell to a destination cell. This alternative code uses the Destination argument to bypass the clipboard and copy the contents of cell A1 directly to cell B1.

```
Range("A1").Copy Destination:=Range("B1")
```

If you only need to copy values (not formatting or formulas), you can improve performance even more by avoiding the Copy method altogether. Simply set the value of the destination cell to the same value found in the source cell. This method is approximately 25 times faster than using the Copy method.

```
Range("B1").Value = Range("A1").Value
```

If you need to copy only the formula from one cell to another (not values or formatting), you can set the formula of the destination cell to the same formula contained in the source cell.

```
Range("B1").Formula = Range("A1").Formula
```

Use the With Statement

When recording macros, you often manipulate the same object more than once. For example, your code may change the formatting of cell A1 so that it is underlined, italicized, and formatted bold. If you were to record a macro when applying these formatting options to cell A1, you would get something like this:

```
Range("A1").Select
Selection.Font.Bold = True
Selection.Font.Italic = True
Selection.Font.Underline = xlUnderlineStyleSingle
```

Unfortunately, this code is not as efficient as it could be because it forces Excel to select and then refer to the object for each change.

You can save time and improve performance by using the With statement to perform several actions on a given object while only referring to the object once. The With statement in this example tells Excel to change three properties, but only reference the Font object one time, which uses less resources:

```
With Range("A1").Font

  .Bold = True
  .Italic = True
  .Underline = xlUnderlineStyleSingle

End With
```

Getting into the habit of chunking actions into With statements not only keeps your macros running faster, but it also helps to make your code easier to read.

Don’t Explicitly Select Objects

If you were to record a macro while entering the value 1000 in cell A1 for multiple sheets, you would end up with code that looks similar to this:

```
Sheets("Sheet1").Select
Range("A1").Select
ActiveCell.FormulaR1C1 = "1000"

Sheets("Sheet2").Select
Range("A1").Select
ActiveCell.FormulaR1C1 = "1000"

Sheets("Sheet3").Select
Range("A1").Select
ActiveCell.FormulaR1C1 = "1000"
```

As you can see, the Macro Recorder is fond of using the Select method to explicitly select objects before taking actions on them. While this code will run fine, it is not all that efficient. It forces Excel to take the time to explicitly select each object being manipulated.

There is generally no need to select objects before working with them. In fact, you can dramatically improve macro performance by not using the Select method.

After recording your macros, make it a habit to alter the generated code to remove the Select methods. In this case, the optimized code would look like this. Note that nothing is being selected. The code simply uses the object hierarchy to apply the needed actions.

```
Sheets("Sheet1").Range("A1").FormulaR1C1 = "1000"
Sheets("Sheet2").Range("A1").FormulaR1C1 = "1000"
Sheets("Sheet3").Range("A1").FormulaR1C1 = "1000"
```

Avoid Excessive Trips to the Worksheet

Another way to speed up your macros is to limit the amount of times you reference worksheet data in your code. It is always less efficient to grab data from the worksheet than from memory. That is to say, your macros run much faster if they do not have to repeatedly interact with the worksheet.

For example, this simple code forces VBA to continuously return to Sheets(“Sheet1”).Range(“A1”) to get the number needed for the comparison being performed in the If statement:

```
For ReportMonth = 1 To 12

  If Range("A1").Value = ReportMonth Then
    MsgBox 1000000 / ReportMonth
  End If

Next ReportMonth
```

A much more efficient way is to save the value in Sheets(“Sheet1”).Range(“A1”) into a variable called MyMonth. This way, the code references the MyMonth variable rather than the worksheet.

```
Dim MyMonth as Integer

MyMonth = Range("A1").Value
```

```
For ReportMonth = 1 To 12
  If MyMonth = ReportMonth Then
    MsgBox 1000000 / ReportMonth
  End If

Next ReportMonth
```

Index

Symbols

& (ampersand), 149
* (asterisk), 95
@ symbol, 145
\ (back slash), 95
: (colon), 95
“ (double quote), 95
/ (forward slash), 80, 95
> (greater than), 95
< (less than), 95
| (pipe), 95
? (question mark), 95
_ (underscore), 35–36

A

absolute references, recording macros with, 14–17
activating
    Edit toolbar (VBE), 238
    Visual Basic (VB) Editor, 4, 12, 29
ActiveCell.PivotTable.Name property, 172, 176, 179, 181, 183
ActiveSheet.PrintOut method, 187–188
ActiveSheet.UsedRange, 150
ActiveX controls, 24
ActiveX Data Objects (ADO), 226–231
Add method, 62, 84
adding
    text to left/right of cells, 147–149
    VBA modules, 32–34
    worksheets, 83–84
addresses, mailing to contact list, 209–211

adjusting pivot data field titles, 172–173
ADO (ActiveX Data Objects), 226–231
aligning charts to specific ranges, 193–195
alphabetical order, sorting fields in, 178–179
ampersand (&), 149
Application object, 44
Application.DisplayStatusBar property, 255
Application.ScreenUpdating property, 254–255
applying
    block comments, 238–239
    custom sort to data items, 179–180
    number formatting to data items, 175–178
    pivot field restrictions, 182–184
    pivot table restrictions, 180–182
ASCII code, 150, 173
assigning macros to buttons and form controls, 22–24
asterisk (*), 95
@ symbol, 145
Atmt variable, 212, 214
attachment
    mailing active workbook as an, 201–203
    mailing single sheets as an, 205–207
    mailing specific ranges as an, 203–205
    saving to folders, 211–213, 213–215
Auto Data Tips option (Editor tab), 39
Auto Indent setting (Editor tab), 39–40
Auto List Members option (Editor tab), 39
Auto Quick Info option (Editor tab), 39
Auto Syntax Check
    option (Editor tab), 38
    turning off, 245–246
AutoFilter function, 155–156, 159
AutoFilterMode property, 155–162, 159
automatic calculations, turning off, 253–254

automating
defined, 1
reporting. *See* reporting
AutoSort method, 179
avoiding Select method, 260

B

back slash (\), 95
backups, creating with today's date, 80
BeforeClose event, 66–68, 78–79
block comments, applying, 238–239
blogs, expert, 250
Boolean variables, 47
BreakLink method, 195
breakpoints, setting in code, 243–244
building
backups of workbooks with today's date, 80
dynamic connections using macros, 222–224
external data connections manually, 218–221
macros, 12
named ranges, 113–116
navigation buttons, 25–26
pivot table inventory summaries, 169–171
sets of disconnected charts, 195–196
table of contents for worksheets, 100–103
workbooks for each report filter item, 189–191
workbooks for each worksheet, 94–96
workbooks from scratch, 61–63
buttons, assigning macros to, 22–24

C

case-sensitivity, of Excel passwords, 68
Cdbl function, 139

cells
adding text to left/right of, 147–149
padding with zeros, 143–145
preventing workbooks from closing until populated, 78–79
replacing blanks with values, 145–147
saving workbooks when cells changed, 63–64
trimming spaces from in ranges, 139–141
Cells statement, 127
Chandoo.org (blog), 250, 251
changing pivot data field titles, 172–173
characters, in filenames, 95
ChartObjects collection, 192–193, 196–197
ChartObjects.ShapeRange.Group, 195–196
charts
aligning to specific ranges, 193–195
creating sets of disconnected, 195–196
printing on worksheets, 196–197
resizing on worksheets, 192–193
Cheat sheet (website), 4
Chr function, 150, 173
classes, training, 251
cleaning up nonprinting characters, 149–151
closing
saving workbooks before, 65–66
workbooks all at once, 76–77
code
ASCII, 150, 173
copying multiple lines of, 239
debugging, 241–242
finding on the Internet, 248–249
stepping through, 241–242
stepping to specific lines, 242–243
stopping at predefined points, 243–244
Code Colors option (Editor Format tab), 41

Code pane (VBE)
about, 31
getting VBA code into modules, 35–37
maximizing, 34–35
minimizing, 34–35
working with, 34–37
collections, 45
colon (:), 95
color, grouping worksheets by, 92–93
columns. *See also* rows
deleting blank, 121–122
displaying in status bar, 158–162
finding blank, 126–128
highlighting, 104–105
performing Text to Columns command on, 134–137
selecting blank, 126–128
unhiding, 119
commenting a line, 13
commenting out code, 238–239
comments, 13
comparing absolute macro recording and relative macro recording, 14–19
components
of macros, 12–13
Visual Basic (VB) Editor, 30–31
connection string, 227
ConnectString argument, 228
contact list, mailing addresses to, 209–211
Contextures (blog), 250
converting
formulas in ranges to values, 132–134
trailing minus signs, 137–139
Copy method, 132, 258–259
CopyFromRecordset method, 230

copying
filtered rows to new workbooks, 17
multiple lines of code, 239
ranges, 131–132
worksheets to new workbooks, 94
copying and pasting, avoiding, 258–259
Count function, 145
COUNTA function, 15
CountIf function, 151–152, 153–154
creating
backups of workbooks with today's date, 80
dynamic connections using macros, 222–224
external data connections manually, 218–221
macros, 12
named ranges, 113–116
navigation buttons, 25–26
pivot table inventory summaries, 169–171
sets of disconnected charts, 195–196
table of contents for worksheets, 100–103
workbooks for each report filter item, 189–191
workbooks for each worksheet, 94–96
workbooks from scratch, 61–63
Ctrl+Page Down, 240
Ctrl+Page Up, 240
Ctrl+R, 240
Ctrl+Tab, 240
CursorType argument, 228
customizing VBE, 38–42

D

DailyDose (blog), 250
Dalgleish, Debra (blogger), 250
data connections

about, 217–218
creating dynamic connections using macros, 222–224
iterating through connections in workbooks, 224–226
manually creating connections, 218–221
manually editing connections, 221–222
pulling external data using ADO and VBA, 226–231
data manipulation
adding text to left /right of cells, 147–149
cleaning up nonprinting characters, 149–151
converting formulas in ranges to values, 132–134
converting trailing minus signs, 137–139
copying and pasting ranges, 131–132
copying filtered rows to new workbooks, 17
displaying filtered columns in status bar, 158–162
hiding rows, 152–154
highlighting duplicates in ranges of data, 151–152
padding cells with zeros, 143–145
performing Text to Columns command on columns, 134–137
replacing blank cells with values, 145–147
selectively hiding AutoFilter drop-down arrows, 155–156
trimming spaces from cells in ranges, 139–141
truncating zip codes to the left five digits, 141–143
Data Source argument, 227
date, creating workbook backups with today's, 80
debugging
stepping to specific lines of code, 242–243
stopping code at predefined points, 243–244
declarations, 32–33
declaring Recordset, 228
Default to Full Module View option (Editor tab), 40
deleting
all but active worksheets, 84–86
blank columns, 121–122
blank rows, 119–121

pivot table drill-down sheets automatically, 184–187
VBA modules, 34
Description (Record Macro dialog box), 11
Developer tab, 10, 29
Dir function, 75, 77
directories
determining if workbooks exist in, 74–76
printing all workbooks in, 77–78
disabling
Auto Syntax Check, 245–246
automatic calculations, 253–254
sheet screen updating, 254–255
status bar updates, 255
.Display method, 205, 206, 209, 211
DisplayAlerts method, 62, 84–85
displaying
Developer tab, 10
filtered columns in status bar, 158–162
DisplayPageBreaks property, 257
dissecting Excel files, 252
Docking tab (VBE), 42
double quote (“), 95
double variables, 47
double-clicks, zooming in/out of worksheets with, 103–104
Drag-and-Drop Text Editing option (Editor tab), 40
DragToColumn property, 182
DragToData property, 182
DragToHide property, 182
DragToPage property, 182
DragToRow property, 182
drill-down sheets, automatically deleting pivot table, 184–187
Duggirala, Purna (blogger), 250
duplicates

hiding rows except ones containing, 152–154
highlighting in ranges of data, 151–152

E

Edit toolbar (VBE)
about, 40
activating, 238
editing
external data connections manually, 221–222
macros, 13
Editor Format tab (VBE), 40–41
Editor tab (VBE), 38–40
emails
saving attachments to folders, 211–213, 213–215
sending active workbook as an attachment, 201–203
sending addresses to contact list, 209–211
sending mail with a link to your workbook, 207–209
sending single sheets as attachments, 205–207
sending specific ranges as attachments, 203–205
EnableDrilldown property, 181, 185
EnableEvents property, 256
EnableFieldDialog property, 181
EnableFieldList property, 181
EnableItemSelection property, 182
EnableWizard property, 181
enabling
Edit toolbar (VBE), 238
Visual Basic (VB) Editor, 4, 12, 29
End property, 127, 128
error handling
about, 52–53
On Error GoTo 0 statement, 55
On Error GoTo SomeLabel statement, 53–54

On Error Resume Next statement, 54
event macros, installing, 60, 82, 110, 130, 166, 200
events
about, 49
BeforeClose, 66–68, 78–79
ignoring, 256
OnAutoFilter, 160
Open, 69–70
workbook, 51–52
Workbook_BeforeClose, 162, 184–185, 187
worksheet, 49–51
Worksheet_Activate, 50, 162
Worksheet_BeforeClose, 51
Worksheet_BeforeDoubleClick, 50, 184–185, 185–186
Worksheet_BeforeRightClick, 510
Worksheet_BeforeSave, 51
Worksheet_Calculate, 51
Worksheet_Change, 50, 256
Worksheet_Deactivate, 51, 162
Worksheet_Open, 51
Worksheet_SelectionChange, 50
Worksheet_SheetChange, 51
Excel, security in, 19–21
Excel events
about, 49
BeforeClose, 66–68, 78–79
ignoring, 256
OnAutoFilter, 160
Open, 69–70
workbook, 51–52
Workbook_BeforeClose, 162, 184–185, 187
worksheet, 49–51
Worksheet_Activate, 50, 162
Worksheet_BeforeClose, 51

Worksheet_BeforeDoubleClick, 50, 184–185, 185–186
Worksheet_BeforeRightClick, 50
Worksheet_BeforeSave, 51
Worksheet_Calculate, 51
Worksheet_Change, 50, 256
Worksheet_Deactivate, 51, 162
Worksheet_Open, 51
Worksheet_SelectionChange, 50
Worksheet_SheetChange, 51
Excel experts, 252
Excel functions
AutoFilter, 155–156, 159
Cdbl, 139
Chr, 150, 173
Count, 145
COUNTA, 15
CountIf, 151–152, 153–154
Dir, 75, 77
Function property, 174
Instr, 215
IsEmpty, 137, 140, 142, 145, 154
IsNumeric, 139
Left, 141
Len, 147
Now, 160
Replace, 185
Right, 143, 145
Sum, 145
teleporting to, 240–241
Trim, 139–141
UCase, 91
Excel Hero Academy (website), 251
Excel methods

about, 46
ActiveSheet.PrintOut, 187–188
Add, 62, 84
AutoSort, 179
BreakLink, 195
Copy, 132, 258–259
CopyFromRecordset, 230
.Display, 205, 206, 209, 211
DisplayAlerts, 62, 84–85
Excel Object Model, 46
GetOpenFilename, 70–72, 74
Hyperlinks.Add, 102
Intersect, 63–64
Move, 88–89
Paste, 62
PasteSpecial, 132
PrintOut, 96
RefreshTable, 168
Replace, 150
Save, 64
SaveAs, 63
SaveCopyAs, 80
Select, 260
ShapeRange.Group, 195
SpecialCells, 125
Excel Object Model
about, 43–44
collections, 45
methods, 46
objects, 44
properties, 45–46
Excel properties
about, 45–46
ActiveCell.PivotTable.Name, 172, 176, 179, 181, 183

Application.DisplayStatusBar, 255
Application.ScreenUpdating, 254–255
AutoFilterMode, 155–162, 159
DisplayPageBreaks, 257
DragToColumn, 182
DragToData, 182
DragToHide, 182
DragToPage, 182
DragToRow, 182
EnableDrilldown, 181, 185
EnableEvents, 256
EnableFieldDialog, 181
EnableFieldList, 181
EnableItemSelection, 182
EnableWizard, 181
End, 127, 128
Function, 174
HasFormula, 134
HTMLBody, 209
Offset, 127, 128, 171
PivotCache.EnableRefresh, 181
PivotTable SourceData, 176–177
PivotTable.ManualUpdate, 257–258
Position, 180
Range, 17
ScreenUpdating, 255
ScrollArea, 122
SourceData, 176, 177
SourceName, 172
StatusBar, 158–160
UsedRange, 119, 121–122
xlSum, 174
Excel variables

about, 46–49
Atmt, 212, 214
beginning values of, 244–245
Boolean, 47
double, 47
end values of, 244–245
FileName, 212, 214
idxCurr, 91, 93
idxPrev, 91, 93
LastColumn, 128
LastRow, 128
long, 47
MItem, 212, 214
MyCell, 117, 136, 140, 142, 144, 146, 148–149, 152, 154
MyInbox, 212, 214
MyMonth, 261
MyRange, 117, 118, 120, 122, 133, 137, 138, 140, 142, 144–145, 146–147, 148–149, 152, 154
Object, 47
OLApp, 202, 205–206, 208, 210
OLMail, 202, 205–206, 208, 210
PivotTable object, 169–171, 172, 177, 179, 181, 182, 183
Require Variable Declaration option (Editor tab), 39
sFormat, 176
sLabel, 176, 177
sPath, 212, 214
SrcRange, 176, 177
String, 47
variant, 47
wbLinks Variant, 196
ExcelGuru (blog), 250
Exceljet (website), 251
exiting debug mode, 242, 244
expert blogs, 250

Extended Properties argument, 227
External Data Connection
about, 217–218
creating dynamic connections using macros, 222–224
iterating through connections in workbooks, 224–226
manually creating connections, 218–221
manually editing connections, 221–222
pulling external data using ADO and VBA, 226–231

F

F5, 37
fields, sorting in alphabetical order, 178–179
file extensions, macro-enabled, 20
FileName variable, 212, 214
filenames, characters in, 95
files, dissecting, 252
Find and Replace functionality, 149
finding
blank rows/columns, 126–128
code on the Internet, 248–249
folders, saving attachments to, 211–213, 213–215
Font option (Editor Format tab), 41
For Each statement, 117, 135, 137–138, 148, 151, 172, 176, 179, 183
form controls, assigning macros to, 22–24
Format Cells dialog box, 143
formatting
formulas in workbooks, 123–125
ranges, 111–113
formulas
converting in ranges to values, 132–134
formatting in workbooks, 123–125
selecting in workbooks, 123–125
forward slash (/), 80, 95

function procedures, 33
Function property, 174
functions
AutoFilter, 155–156, 159
Cdbl, 139
Chr, 150, 172
Count, 145
COUNTA, 15
CountIf, 151–152, 153–154
Dir, 75, 77
Function property, 174
Instr, 215
IsEmpty, 137, 140, 145, 154
IsNumeric, 139
Left, 141
Len, 147
Now, 160
Replace, 185
Right, 143, 145
Sum, 145
teleporting to, 240–241
Trim, 139–141
UCase, 91

G

General tab (VBE), 41–42
GetOpenFilename method, 70–72, 74
Go To Special dialog box, 124
greater than (>), 95
grouping worksheets by color, 92–93

H

HasFormula property, 134

help, 247–252
hiding
all but active worksheets, 86–87
AutoFilter drop-down arrows, 155–156
page breaks, 257
rows, 152–154
highlighting
active rows/columns, 104–105
duplicates in ranges of data, 151–152
HTMLBody property, 209
Hyperlinks.Add method, 102

I

icons, explained, 3
idxCurr variable, 91, 93
idxPrev variable, 91, 93
If statement, 64
ignoring events, 256
immediate window (VBE), 31
Import Data dialog box, 219–220
importing text files to ranges, 233–234
Input statement, 231–232
Input # statement, 232
inserting blank rows in ranges, 117–118
installing
event macros, 60, 82, 110, 130, 166, 200
macros, 60–61, 81–83, 110–111, 130–131, 166–167, 200–201
Personal Macro Workbook, 60–61, 110–111, 131, 166–167, 200–201
Personal Mcro Workbook, 82–83
standard macros, 61, 83, 111, 131, 167, 201
Instr function, 215
Internet, finding code on the, 248–249
Internet resources. *See also* blogs, expert

Chandoo.org, 251
Cheat sheet, 4
Excel Hero Academy, 251
Exceljet, 251
Microsoft Office Dev Center, 251
sample files, 3–4
YouTube, 251
Intersect method, 63–64
IsEmpty function, 137, 140, 145, 154
IsNumeric function, 139
iterating, through connections in workbooks, 224–226

J

Jelen, Bill (blogger), 250
jumping between modules and procedures, 240

K

Kusleika, Dick (blogger), 250

L

LastColumn variable, 128
LastRow variable, 128
Left function, 141
Len function, 147
less than (<), 95
limiting movement of ranges, 122–123
Line Input # statement, 232
line-continuation character, 36
links, sending for workbooks, 207–209
live training classes, 251
LockType argument, 228
long variables, 47
looping, through ranges, 116–117

M

Macro Name (Record Macro dialog box), 11
Macro Recorder, help with, 248
macro-enabled file extensions, 20
macros. *See also specific topics*
- assigning to buttons and form controls, 22–24
- components of, 12–13
- creating, 12
- creating dynamic connections using, 222–224
- defined, 1, 9
- editing, 13
- error handling, 52–55
- event procedures, 49–52
- examples of, 25–28
- Excel Object Model, 43–46
- help with, 247–252
- installing, 60–61, 81–83, 110–111, 130–131, 166–167, 200–201
- manipulating data with. *See* data manipulation
- placing on Quick Access toolbar, 24–25
- reasons for using, 9–10
- recording, 9, 10–25
- recording with absolute references, 14–17
- recording with relative references, 17–19
- RefreshQuery, 223
- speeding up, 253–261
- storing in personal macro workbook, 22
- testing, 14
- variables, 46–49

manipulating data
- adding text to left /right of cells, 147–149
- cleaning up nonprinting characters, 149–151
- converting formulas in ranges to values, 132–134
- converting trailing minus signs, 137–139

copying and pasting ranges, 131–132
copying filtered rows to new workbooks, 17
displaying filtered columns in status bar, 158–162
hiding rows, 152–154
highlighting duplicates in ranges of data, 151–152
padding cells with zeros, 143–145
performing Text to Columns command on columns, 134–137
replacing blank cells with values, 145–147
selectively hiding AutoFilter drop-down arrows, 155–156
trimming spaces from cells in ranges, 139–141
truncating zip codes to the left five digits, 141–143
MAPI namespace, 212, 214
Margin Indicator Bar option (Editor Format tab), 41
maximizing Code pane (VBE), 34–35
Menu bar (VBE), 30
methods
about, 46
ActiveSheet.PrintOut, 187–188
Add, 62, 84
AutoSort, 179
BreakLink, 195
Copy, 132, 258–259
CopyFromRecordset, 230
.Display, 205, 206, 209, 211
DisplayAlerts, 62, 84–85
Excel Object Model, 46
GetOpenFilename, 70–72, 74
Hyperlinks.Add, 102
Intersect, 63–64
Move, 88–89
Paste, 62
PasteSpecial, 132
PrintOut, 96
RefreshTable, 168

Replace, 150
Save, 64
SaveAs, 63
SaveCopyAs, 80
Select, 260
ShapeRange.Group, 195
SpecialCells, 125
Microsoft Office Dev Center, 251
Microsoft Outlook Object Library, 202, 203, 205, 206, 210, 213
minimizing Code pane (VBE), 34–35
mistakes, using macros for avoiding, 10
MItem variable, 212, 214
modules, jumping between procedures and, 240
Move method, 88–89
moving worksheets, 88–89
MrExcel (blog), 250
MyCell variable, 117, 136, 140, 142, 144, 146, 148–149, 152, 154
MyInbox variable, 212, 214
MyMonth variable, 261
MyRange variable, 117, 118, 120, 122, 133, 137, 138, 140, 142, 144–145, 146–147, 148–149, 152, 154

N

name, sorting worksheets by, 90–91
Name Manager command, 114
naming worksheets, 83–84
navigation
building buttons for, 25–26
using macros for, 10
nonprinting characters, cleaning up, 149–151
Now function, 160
number formatting, applying to data items, 175–178

O

Object variable, 47
objects
about, 44
Application, 44
PageField, 187, 189, 191
PivotItems, 180
PivotTable, 169–171, 172, 177, 179, 181, 182, 183, 189
ThisWorkbook, 168
Offset property, 127, 128, 171
OLApp variable, 202, 205–206, 208, 210
OLApp.Session.Logon, 206, 208
OLMail variable, 202, 205–206, 208, 210
On Error GoTo 0 statement, 53, 55
On Error GoTo SomeLabel statement, 53–54
On Error Resume Next statement, 53, 54, 176, 179, 190, 212, 215
OnAutoFilter event, 160
one-touch reporting options, 27–28
online training classes, 251
Open event, 69–70
Open statement, 231–232
opening
text files, 231–232
workbooks defined by users, 70–72
workbooks to specific tabs, 69–70

P

padding cells with zeros, 143–145
page breaks, hiding, 257
PageField object, 187, 189, 191
Password argument, 97–98, 227
passwords, case-sensitivity of, 68
Paste method, 62

PasteSpecial method, 132
pasting
copying and, 258–259
ranges, 131–132
Personal Macro Workbook
installing, 60–61, 82–83, 110–111, 131, 166–167, 200–201
storing macros in, 22
pipe (|), 95
pivot cache, 171
Pivot Table object, 189
pivot tables
adjusting data field titles, 172–173
applying restrictions, 180–182
applying restrictions to pivot fields, 182–184
automatically deleting drill-down sheets, 184–187
creating inventory summaries, 169–171
printing for each report filter item, 187–189
rearranging data, 26–27
refreshing in workbooks, 167–168
suspending updates, 257–258
PivotCache.EnableRefresh property, 181
PivotItems object, 180
PivotTable object variable, 169–171, 172, 177, 179, 181, 182, 183
PivotTable SourceData property, 176–177
PivotTable.ManualUpdate property, 257–258
placing macros on Quick Access toolbar, 24–25
Position property, 180
preventing workbooks from closing until cells are populated, 78–79
Print # statement, 232
printing
all workbooks in directories, 77–78
charts on worksheets, 196–197
pivot tables for each report filter item, 187–189
specified worksheets, 96–97

PrintOut method, 96
Procedure Separator option (Editor tab), 40
procedures
    jumping between modules and, 240
    staying with one, 241
Project Explorer (VBE)
    adding modules, 32–34
    working with, 31–34
properties
    about, 45–46
    ActiveCell.PivotTable.Name, 172, 176, 179, 181, 183
    Application.DisplayStatusBar, 255
    Application.ScreenUpdating, 254–255
    AutoFilterMode, 155–162, 159
    DisplayPageBreaks, 257
    DragToColumn, 182
    DragToData, 182
    DragToHide, 182
    DragToPage, 182
    DragToRow, 182
    EnableDrilldown, 181, 185
    EnableEvents, 256
    EnableFieldDialog, 181
    EnableFieldList, 181
    EnableItemSelection, 182
    EnableWizard, 181
    End, 128, 135
    Function, 174
    HasFormula, 134
    HTMLBody, 209
    Offset, 127, 128, 171
    PivotCache.EnableRefresh, 181
    PivotTable SourceData, 176–177

PivotTable.ManualUpdate, 257–258
Position, 180
Range, 17
ScreenUpdating, 255
ScrollArea, 122
SourceData, 176, 177
SourceName, 172
StatusBar, 158–160
UsedRange, 119, 121–122
xlSum, 174
protecting worksheets, 67–68, 97–98
Provider argument, 227
Puls, Ken (blogger), 250

Q

question mark (?), 95
Quick Access toolbar, placing macros on, 24–25

R

Range property, 17
ranges
about, 109–110
aligning charts to specific, 193–195
converting formulas in to values, 132–134
copying, 131–132
creating named, 113–116
deleting blank columns, 121–122
deleting blank rows, 119–121
finding blank rows/columns, 126–128
formatting, 111–113
formatting formulas in workbooks, 123–125
highlighting duplicate in data, 151–152
importing text files to, 233–234

inserting blank rows in, 117–118
limiting movement of, 122–123
looping through, 116–117
mailing specific ranges as attachments, 203–205
pasting, 131–132
selecting, 111–113
selecting blank rows/columns, 126–128
selecting formulas in workbooks, 123–125
selecting named, 113–116
trimming spaces from cells in, 139–141
unhiding all rows/columns, 119
reading opened text files, 232
rearranging pivot table data, 26–27
Record Macro dialog box, 10–11
recording
macros, 9, 10–25
macros with absolute references, 14–17
macros with relative references, 17–19
Recordset, declaring, 228
References dialog box, 229
referencing worksheet data, 261
refreshing pivot tables in workbooks, 167–168
RefreshQuery macro, 223
RefreshTable method, 168
relative macro recording, compared with absolute macro recording, 14–19
relative references, recording macros with, 17–19
Remember icon, 3
removing
all but active worksheets, 84–86
blank columns, 121–122
blank rows, 119–121
pivot table drill-down sheets automatically, 184–187
VBA modules, 34
repetitive tasks, using macros for, 9–10

Replace function, 185
Replace method, 150
replacing blank cells with values, 145–147
report filter
    creating workbooks for each, 189–191
    printing pivot tables for each, 187–189
reporting
    adjusting pivot data field titles, 172–173
    aligning charts to specific ranges, 193–195
    applying custom sort to data items, 179–180
    applying number formatting to data items, 175–178
    applying pivot field restrictions, 182–184
    applying pivot table restrictions, 180–182
    automatically deleting pivot table drill-down sheets, 184–187
    creating pivot table inventory summaries, 169–171
    creating sets of disconnected charts, 195–196
    creating workbooks for each report filter item, 189–191
    one-touch, 27–28
    printing charts on worksheets, 196–197
    printing pivot tables for each report filter item, 187–189
    refreshing pivot tables in workbooks, 167–168
    resizing charts on worksheets, 192–193
    setting data items to sum, 173–175
    sorting fields in alphabetical order, 178–179
Require Variable Declaration option (Editor tab), 39
resizing charts on worksheets, 192–193
resources. *See also* blogs, expert
    Chandoo.org, 251
    Cheat sheet, 4
    Excel Hero Academy, 251
    Exceljet, 251
    Microsoft Office Dev Center, 251
    sample files, 3–4

YouTube, 251
restrictions
applying pivot field, 182–184
applying pivot table, 180–182
Right function, 143, 145
rows. *See also* columns
copying to new workbooks, 17
deleting blank, 119–121
finding blank, 126–128
hiding, 152–154
highlighting, 104–105
inserting blanks in ranges, 117–118
selecting blank, 126–128
unhiding, 119

S

sample files (website), 3–4
Save method, 64
SaveAs method, 63
SaveCopyAs method, 80
saving
attachments to folders, 211–213, 213–215
workbooks before closing, 65–66
workbooks when cells changed, 63–64
ScreenUpdating property, 255
ScrollArea property, 122
security, in Excel, 19–21
Select Case statement, 65–66
Select Data Source dialog box, 219–220
Select method, 260
Select Table dialog box, 219–220
selecting
blank rows/columns, 126–128

formulas in workbooks, 123–125
named ranges, 113–116
ranges, 111–113
sending
emails. *See* emails
mail with a link to your workbook, 207–209
SendMail command, 201
setting
breakpoints in code, 243–244
data items to sum, 173–175
setup, of trusted locations, 20–21
sFormat variable, 176
ShapeRange.Group method, 195
sheet screen, disabling updating, 254–255
Shortcut Key (Record Macro dialog box), 11
Size setting (Editor Format tab), 41
sLabel variable, 176, 177
sorting
applying custom sort to data items, 179–180
fields in alphabetical order, 178–179
worksheets by name, 90–91
Source argument, 228
SourceData property, 176, 177
SourceName property, 172
spaces, trimming from cells in ranges, 139–141
sPath variable, 212, 214
SpecialCells method, 125
speed, of macros, 253–261
SQL (Structured Query Language), 221–222
SQL statement, 222–223
SrcRange variable, 176, 177
standard macros, installing, 61, 83, 111, 131, 167, 201
standard toolbar (VBE), 31
statements

With, 259–260
about, 35–36
Cells, 127
For Each, 117, 135, 137–138, 148, 151, 172, 176, 179, 183
On Error GoTo 0, 53, 55
On Error GoTo SomeLabel, 53–54
On Error Resume Next, 53, 54, 176, 179, 190, 212, 215
If, 64
Input, 231–232
Input #, 232
Line Input #, 232
Open, 231–232
Print #, 232
Select Case, 65–66
SQL, 222–223
With-End With, 112, 125
Write #, 232
status bar
displaying filtered columns in, 158–162
turning off updates, 255
StatusBar property, 158–160
stepping through
code, 241–242
to specific code lines, 242–243
stopping
code at predefined points, 243–244
pivot table updates, 257–258
worksheet calculations, 253–254
Store Macro In (Record Macro dialog box), 11
storing macros in personal macro workbook, 22
String variable, 47
Structured Query Language (SQL), 221–222
sub procedures, 33

Sum function, 145

T

Tab key, 39
table of contents, creating for worksheets, 100–103
tables, pivot
    adjusting data field titles, 172–173
    applying restrictions, 180–182
    applying restrictions to pivot fields, 182–184
    automatically deleting drill-down sheets, 184–187
    creating inventory summaries, 169–171
    printing for each report filter item, 187–189
    rearranging data, 26–27
    refreshing in workbooks, 167–168
    suspending updates, 257–258
tabs, opening workbooks to specific, 69–70
Technical Stuff icon, 3
teleporting, to functions, 240–241
testing macros, 14
text, adding to left/right of cells, 147–149
text files
    importing to ranges, 233–234
    logging workbook usage in, 232–233
    opening, 231–232
    reading, 232
Text to Columns command, performing on columns, 134–137
Text to Columns Wizard dialog box, 134
ThisWorkbook object, 168
Tip icon, 3
toolbar (VBE), 31
trailing minus signs, converting, 137–139
training classes, 251
Trim function, 139–141

trimming spaces from cells in ranges, 139–141
truncating zip codes to the left five digits, 141–143
trusted document, 20
trusted locations, 20–21
turning off
    Auto Syntax Check, 245–246
    automatic calculations, 253–254
    sheet screen updating, 254–255
    status bar updates, 255
turning on
    Edit toolbar (VBE), 238
    Visual Basic (VB) Editor, 4, 12, 29

U

UCase function, 91
underscore (_), 35–36
Unhide dialog box, 87
unhiding
    all worksheets in a workbook, 87–88
    rows/columns, 119
unprotecting worksheets, 68–69, 98–100
updating
    disabling sheet screen, 254–255
    suspending pivot table updates, 257–258
    turning off status bar updates, 255
UsedRange property, 119, 121–122
user forums, 249–250
User ID argument, 227
users, opening workbooks defined by, 70–72

V

values
    converting formulas in ranges to, 132–134

replacing blank cells with, 145–147
of variables, 244–245
variables
about, 46–49
Atmt, 212, 214
beginning values of, 244–245
Boolean, 47
double, 47
end values of, 244–245
FileName, 212, 214
idxCurr, 91, 93
idxPrev, 91, 93
LastColumn, 128
LastRow, 128
long, 47
MItem, 212, 214
MyCell, 117, 136, 140, 142, 144, 146, 148–149, 152, 154
MyInbox, 212, 214
MyMonth, 261
MyRange, 117, 118, 120, 122, 133, 137, 138, 140, 142, 144–145, 146–147, 148–149, 152, 154
Object, 47
OLApp, 202, 205–206, 208, 210
OLMail, 202, 205–206, 208, 210
PivotTable object, 169–171, 172, 177, 179, 181, 182, 183
Require Variable Declaration option (Editor tab), 39
sFormat, 176
sLabel, 176, 177
sPath, 212, 214
SrcRange, 176, 177
String, 47
variant, 47
wbLinks Variant, 196
variant variables, 47

VBA (Visual Basic for Applications)
about, 1
adding modules, 32–34
getting code into modules, 35–37
help files, 248
pulling external data using, 226–231
removing modules, 34
VBA statements
With, 259–260
about, 35–36
Cells, 127
For Each, 117, 135, 137–138, 148, 151, 172, 176, 179, 183
On Error GoTo 0, 53, 55
On Error GoTo SomeLabel, 53–54
On Error Resume Next, 53, 54, 176, 179, 190, 212, 215
If, 64
Input, 231–232
Input #, 232
Line Input #, 232
Open, 231–232
Print #, 232
Select Case, 65–66
SQL, 222–223
With-End With, 112, 125
Write #, 232
VBE (Visual Basic Editor)
activating, 4, 12, 29
Code pane, 34–37
components, 30–31
customizing environment, 38–42
Docking tab, 42
Editor Format tab, 40–41
Editor tab, 38–40
General tab, 41–42

Project Explorer, 31–34
removing VBA modules, 34
tips for, 237–246
working in, 29–31
video training, on YouTube, 251
viewing
beginning variable values, 244–245
ending variable values, 244–245
Visual Basic Editor (VBE)
activating, 4, 12, 29
Code pane, 34–37
components, 30–31
customizing environment, 38–42
Docking tab, 42
Editor Format tab, 40–41
Editor tab, 38–40
General tab, 41–42
Project Explorer, 31–34
removing VBA modules, 34
tips for, 237–246
working in, 29–31
Visual Basic for Applications (VBA)
about, 1
adding modules, 32–34
getting code into modules, 35–37
help files, 248
pulling external data using, 226–231
removing modules, 34

W

Warning! icon, 3
wbLinks Variant variable, 196
websites. *See also* blogs, expert

Chandoo.org, 251
Cheat sheet, 4
Excel Hero Academy, 251
Exceljet, 251
Microsoft Office Dev Center, 251
sample files, 3–4
YouTube, 251
With statement, 259–260
With-End With statement, 112, 125
workbook events, 51–52
Workbook_BeforeClose event, 162, 184–185, 187
workbooks
closing all at once, 76–77
copying filtered rows to new, 17
copying worksheets to new, 94
creating backups with today's date, 80
creating for each report filter item, 189–191
creating for each worksheet, 94–96
creating from scratch, 61–63
determining if exists in directory, 74–76
determining if open, 72–74
formatting formulas in, 123–125
iterating through connections in, 224–226
logging usage of in text files, 232–233
mailing active workbook as an attachment, 201–203
opening defined by users, 70–72
opening to specific tabs, 69–70
preventing from closing until cells are populated, 78–79
printing all in directories, 77–78
protecting worksheets on Workbook Close, 67–68
refreshing pivot tables in, 167–168
saving before closing, 65–66
saving when cells changed, 63–64
selecting formulas in, 123–125

sending mail with a link to your, 207–209
unhiding all worksheets in, 87–88
unprotecting worksheets on Workbook Open, 68–69
worksheet events, 49–51
Worksheet_Activate, 50, 162
Worksheet_BeforeClose, 51
Worksheet_BeforeDoubleClick, 50, 184–185, 185–186
Worksheet_BeforeRightClick, 50
Worksheet_BeforeSave, 51
Worksheet_Calculate, 51
Worksheet_Change, 50, 256
Worksheet_Deactivate, 51, 162
Worksheet_Open, 51
worksheets
adding, 83–84
copying to new workbooks, 94
creating table of contents for, 100–103
creating workbooks for each, 94–96
deleting all but active, 84–86
grouping by color, 92–93
hiding all but active, 86–87
highlighting active rows/columns, 104–105
mailing as attachments, 205–207
moving, 88–89
naming, 83–84
printing charts on, 196–197
printing specified, 96–97
protecting, 97–98
protecting on Workbook Close, 67–68
referencing data in, 261
resizing charts on, 192–193
sorting by name, 90–91
stopping calculations, 253–254

unhiding all, 87–88
unprotecting, 98–100
unprotecting on Workbook Open, 68–69
zooming in/out of with double-clicks, 103–104
Worksheet_SelectionChange, 50
Worksheet_SheetChange, 51
Write # statement, 232

X

xlSheetHidden, 87
xlSheetVeryHidden, 87
xlSheetVisible, 88
.xlsm file extension, 20
xlSum property, 174
.xlsx file format, 20

Y

YouTube, video training on, 251

Z

zeros, padding cells with, 143–145
zip codes, truncating to the left five digits, 141–143
zooming in/out, of worksheets with double-click, 103–104

Dedication

To Marlynda

Author's Acknowledgments

Thanks to the team at Wiley for their tireless efforts in completing this book. Thanks to Guy Hart-Davis for his tremendous technical contributions. And special thanks to Rebecca Senninger whose professionalism, patience, and kindness made this book so much better.

About the Author

Dick Kusleika has been working with Microsoft Office for more than 20 years. He was formerly a Microsoft MVP, having been awarded 12 consecutive years. Dick has written several books about Excel and Access.

Publisher's Acknowledgments

Associate Editor: Elizabeth Stilwell

Development Editor: Rebecca Senninger

Technical Editor: Guy Hart-Davis

Production Editor: Saikarthick Kumarasamy

Cover Image: Andrey Popov/Shutterstock

Take dummies with you everywhere you go!

Whether you are excited about e-books, want more from the web, must have your mobile apps, or are swept up in social media, dummies makes everything easier.



Find us online!

dummies.com

dummies
A Wiley Brand

Leverage the power

Dummies is the global leader in the reference category and one of the most trusted and highly regarded brands in the world. No longer just focused on books, customers now have access to the dummies content they need in the format they want. Together we'll craft a solution that engages your customers, stands out from the competition, and helps you meet your goals.

Advertising & Sponsorships

Connect with an engaged audience on a powerful multimedia site, and position your message alongside expert how-to content. Dummies.com is a one-stop shop for free, online information and know-how curated by a team of experts.

- Targeted ads
- Video
- Email Marketing
- Microsites
- Sweepstakes sponsorship



20 MILLION PAGE VIEWS EVERY SINGLE MONTH

15 MILLION UNIQUE VISITORS PER MONTH

43% OF ALL VISITORS ACCESS THE SITE VIA THEIR MOBILE DEVICES

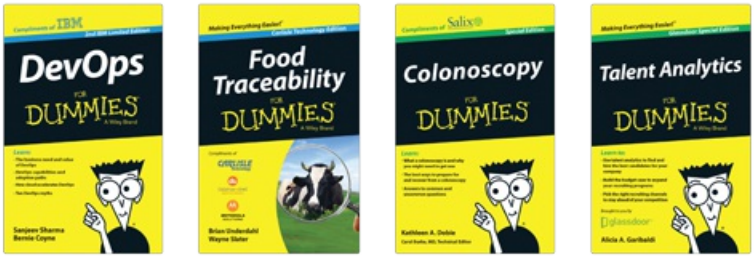
700,000 NEWSLETTER SUBSCRIPTIONS TO THE INBOXES OF *300,000* UNIQUE INDIVIDUALS EVERY WEEK

# of dummies

Custom Publishing

Reach a global audience in any language by creating a solution that will differentiate you from competitors, amplify your message, and encourage customers to make a buying decision.

- Apps
- Books
- eBooks
- Video
- Audio
- Webinars



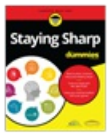
Brand Licensing & Content

Leverage the strength of the world's most popular reference brand to reach new audiences and channels of distribution.

For more information, visit dummies.com/biz

dummies
A Wiley Brand

PERSONAL ENRICHMENT



9781119187790
USA $26.00
CAN $31.99
UK £19.99



9781119179030
USA $21.99
CAN $25.99
UK £16.99



9781119293354
USA $24.99
CAN $29.99
UK £17.99



9781119293347
USA $22.99
CAN $27.99
UK £16.99



9781119310068
USA $22.99
CAN $27.99
UK £16.99



9781119235606
USA $24.99
CAN $29.99
UK £17.99



9781119251163
USA $24.99
CAN $29.99
UK £17.99



9781119235491
USA $26.99
CAN $31.99
UK £19.99



9781119279952
USA $24.99
CAN $29.99
UK £17.99



9781119283133
USA $24.99
CAN $29.99
UK £17.99



9781119287117
USA $24.99
CAN $29.99
UK £16.99

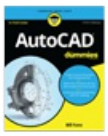


9781119130246
USA $22.99
CAN $27.99
UK £16.99

PROFESSIONAL DEVELOPMENT



9781119311041
USA $24.99
CAN $29.99
UK £17.99



9781119255796
USA $39.99
CAN $47.99
UK £27.99



9781119293439
USA $26.99
CAN $31.99
UK £19.99



9781119281467
USA $26.99
CAN $31.99
UK £19.99



9781119280651
USA $29.99
CAN $35.99
UK £21.99



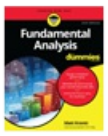
9781119251132
USA $24.99
CAN $29.99
UK £17.99



9781119310563
USA $34.00
CAN $41.99
UK £24.99



9781119181705
USA $29.99
CAN $35.99
UK £21.99



9781119263593
USA $26.99
CAN $31.99
UK £19.99



9781119257769
USA $29.99
CAN $35.99
UK £21.99



9781119293477
USA $26.99
CAN $31.99
UK £19.99



9781119265313
USA $24.99
CAN $29.99
UK £17.99



9781119239314
USA $29.99
CAN $35.99
UK £21.99



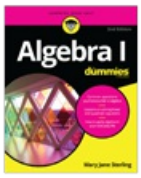
9781119293323
USA $29.99
CAN $35.99
UK £21.99

dummies.com

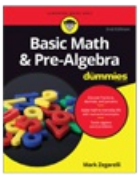
dummies
A Wiley Brand

Learning Made Easy

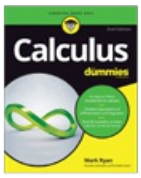
ACADEMIC



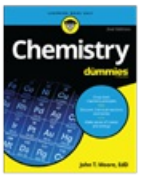
9781119293576
USA $19.99
CAN $23.99
UK £15.99



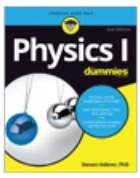
9781119293637
USA $19.99
CAN $23.99
UK £15.99



9781119293491
USA $19.99
CAN $23.99
UK £15.99



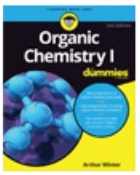
9781119293460
USA $19.99
CAN $23.99
UK £15.99



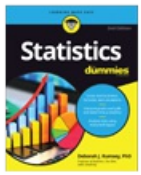
9781119293590
USA $19.99
CAN $23.99
UK £15.99



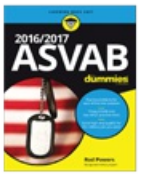
9781119215844
USA $26.99
CAN $31.99
UK £19.99



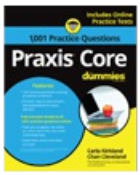
9781119293378
USA $22.99
CAN $27.99
UK £16.99



9781119293521
USA $19.99
CAN $23.99
UK £15.99



9781119239178
USA $18.99
CAN $22.99
UK £14.99



9781119263883
USA $26.99
CAN $31.99
UK £19.99

Available Everywhere Books Are Sold

dummies.com

dummies
A Wiley Brand

Small books for big imaginations



9781119177173
USA $9.99
CAN $9.99
UK £8.99



9781119177272
USA $9.99
CAN $9.99
UK £8.99



9781119177241
USA $9.99
CAN $9.99
UK £8.99



9781119177210
USA $9.99
CAN $9.99
UK £8.99



9781119262657
USA $9.99
CAN $9.99
UK £6.99



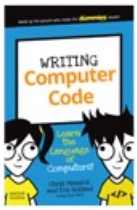
9781119291336
USA $9.99
CAN $9.99
UK £6.99



9781119233527
USA $9.99
CAN $9.99
UK £6.99



9781119291220
USA $9.99
CAN $9.99
UK £6.99



9781119177302
USA $9.99
CAN $9.99
UK £8.99

Unleash Their Creativity

dummies.com

dummies
A Wiley Brand

Take Dummies with you everywhere you go!



Go to our Website



Like us on Facebook



Follow us on Twitter



Watch us on YouTube



Join us on LinkedIn



Pin us on Pinterest



Subscribe to our newsletter



Create your own Dummies book cover


for
dummies
A Wiley Brand

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.