

Microsoft®



Step by Step

Microsoft

Office Excel 2007

Visual Basic® for Applications

Build *exactly* the skills you need.
Learn at the pace *you* want.

Reed Jacobson

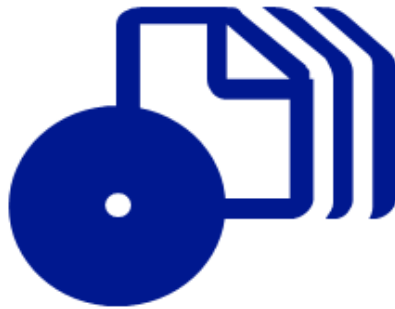


Easy-search CD includes:

- Skill-building practice files
- Complete eBook



How to access your CD files



The print edition of this book includes a CD. To access the CD files, go to <http://aka.ms/624023/files>, and look for the Downloads tab.

Note: Use a desktop web browser, as files may not be accessible from all ereader devices.

Questions? Please contact: mspinput@microsoft.com

Microsoft®

**Microsoft® Office
Excel® 2007
Visual Basic®
for Applications
Step by Step**

Reed Jacobson

PUBLISHED BY
Microsoft Press
A Division of Microsoft Corporation
One Microsoft Way
Redmond, Washington 98052-6399

Copyright © 2007 by Reed Jacobson

All rights reserved. No part of the contents of this book may be reproduced or transmitted in any form or by any means without the written permission of the publisher.

Library of Congress Control Number: 2007924651

ISBN: 978-0-7356-2402-3

Printed and bound in the United States of America.

6 7 8 9 10 11 12 13 14 QGT 7 6 5 4 3 2

Distributed in Canada by H.B. Fenn and Company Ltd.

A CIP catalogue record for this book is available from the British Library.

Microsoft Press books are available through booksellers and distributors worldwide. For further information about international editions, contact your local Microsoft Corporation office or contact Microsoft Press International directly at fax (425) 936-7329. Visit our Web site at www.microsoft.com/mspress. Send comments to mspinput@microsoft.com.

Microsoft, Microsoft Press, ActiveX, Calibri, Excel, Groove, InfoPath, Internet Explorer, MS-DOS, OneNote, Outlook, PivotTable, PowerPoint, SharePoint, SQL Server, Visio, Visual Basic, Visual Studio, Win32, Windows, Windows Server, and Windows Vista are either registered trademarks or trademarks of Microsoft Corporation in the United States and/or other countries. Other product and company names mentioned herein may be the trademarks of their respective owners.

The example companies, organizations, products, domain names, e-mail addresses, logos, people, places, and events depicted herein are fictitious. No association with any real company, organization, product, domain name, e-mail address, logo, person, place, or event is intended or should be inferred.

This book expresses the author's views and opinions. The information contained in this book is provided without any express, statutory, or implied warranties. Neither the authors, Microsoft Corporation, nor its resellers, or distributors will be held liable for any damages caused or alleged to be caused either directly or indirectly by this book.

Acquisitions Editor: Juliana Aldous Atkinson

Developmental Editor: Sandra Haynes

Project Editor: Rosemary Caperton

Editorial and Production Services: Online Training Solutions, Inc.

Technical Reviewer: Jason Lee; Technical Review services provided by Content Master, a member of CM Group, Ltd

Body Part No. X13-68402

[2012-05-04]

Contents

About the Author	xi
Features and Conventions of This Book	xiii
Using the Book's CD	xv
What's on the CD?	xv
Minimum System Requirements	xvi
Step-by-Step Exercises	xvi
2007 Microsoft Office System	xvi
Installing the Practice Files	xviii
Using the Practice Files	xix
Removing and Uninstalling the Practice Files	xx
Getting Help	xxi
Errata & Book Support	xxi

1	Make a Macro Do Simple Tasks	1
	What's the Difference Between VBA and a Macro?	2
	Sidebar: VBA and the .NET Framework	4
	Creating a Simple Macro	5
	Format Currency by Using a Built-In Tool	5
	Record a Macro to Format Currency	6
	Run the Macro	8
	Assign a Shortcut Key to the Macro	8
	Look at the Macro	9
	Save the Macro Workbook	12

 **What do you think of this book? We want to hear from you!**

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

Changing Multiple Properties at Once	13
Create Sidebar Headings with a Command	14
Record a Macro to Merge Cells Vertically	15
Eliminate Unnecessary Lines from the Macro	17
Manipulating Recorded Properties	18
Record a Macro to Remove Window Elements	18
Run the Macro from the Visual Basic Editor	19
Use a Macro to Toggle the Value of a Property	20
Eliminate Repeated Objects in a Recorded Macro	21
Run a Macro from the Quick Access Toolbar	22
Recording Methods in a Macro	24
Convert a Formula to a Value by Using Menu Commands	24
Convert a Formula to a Value by Using a Macro	26
Make a Long Statement More Readable	28
Trusting Macro-Enabled Workbooks	28
Designate a Trusted Location for Macros	29
Designate a Trusted Publisher for Macros	30
Key Points	35

2 Make a Macro Do Complex Tasks 37

Task One: Opening the Report File	39
Open a Text File	39
Watch a Macro Run by Stepping Through It	42
Select a File While Running a Macro	45
Task Two: Filling In Missing Labels	47
Select Only the Blank Cells	48
Fill the Selection with Values	49
Record Filling In the Missing Values	50
Watch the FillLabels Macro Run	50
Task Three: Adding a Column of Dates	52
Add a Constant Date	52
Step Through the Macro	52
Prompt for the Date	53
Task Four: Appending to the Database	55
Append Data to a Master List	55
Step Through the AppendData Macro	56

Record a Relative Movement	58
Choose Whether to Save Changes While Closing a File.	60
Task Five: Deleting the Worksheet	61
Create a Macro to Delete the Active Worksheet.	61
Make the Macro Operate Quietly	62
Assembling the Pieces.	63
Record a Macro That Runs Other Macros	63
Simplify the Subroutine Statements	64
Key Points	65
3 Explore Workbooks and Worksheets	67
What Is an Object?.	68
Objects Come in Collections	68
Objects Have Properties	69
Objects Have Methods	70
Methods Can Change Properties.	72
Properties Can Involve Actions	72
Understanding Workbooks	73
Add a New Workbook.	73
Sidebar: Dockable Views.	76
Count the Workbooks.	77
Close the Workbooks	78
Refer to a Single Workbook.	79
Refer to a Workbook by Name	81
Refer to a Workbook by Pointing	82
Change a Workbook Property Value	83
Understanding Worksheets	84
Add a New Worksheet	84
Rename and Delete a Worksheet	84
Look at the Return Value of the Delete Method.	85
Look at the Result of the Add Method.	86
Copy a Worksheet	87
Manipulate Multiple Worksheets.	87
Declare Variables to Enable Auto Lists	89
Key Points	91

4	Explore Range Objects	93
	Referring to a Range	94
	Refer to a Range by Using an Address	94
	Refer to a Range as a Collection of Cells	97
	Refer to a Range as a Collection of Rows or Columns	101
	Refer to a Range Based on the Active Cell	104
	Refer to Subsets of a Range	107
	Refer to a Relative Range	109
	Enhancing Recorded Selections.	113
	Simplify Select...Selection Pairs	113
	Simplify Select Groups	115
	Entering Values and Formulas into a Range	116
	Relative References	116
	Absolute References	117
	R1C1 Reference Style.	119
	Put Values and Formulas into a Range.	120
	Construct Formulas to Fill a Grid.	123
	Formatting a Range.	127
	Add Borders to a Range	127
	Format the Interior of a Range	131
	Key Points	135
5	Explore Data Objects	137
	Working with Excel Tables	138
	Create a New File from an Existing Worksheet.	138
	Create a Table from an Internal Source	140
	Create a Table from an External Source	142
	Record a Macro to Manipulate a Table.	146
	Manipulate Table Columns.	148
	Manipulate Table Totals and Filters.	151
	Working with PivotTable Reports	153
	Create a PivotTable Report from an Internal Source	153
	Create a PivotTable Report from an External Source.	156
	Record a Macro to Set the PivotTable Structure	158
	Set the PivotTable Structure	160
	Record a Macro to Customize a PivotTable Layout	162
	Customize a PivotTable Layout	164

	Record a Macro to Customize a PivotTable Style	167
	Customize a PivotTable Style	170
	Key Points	173
6	Explore Graphical Objects	175
	Exploring Graphical Objects.	176
	Use Worksheet Cells as a Drawing Grid	176
	Add a Gradient Fill to a Cell	178
	Add a Gradient-Filled Shape	182
	Reference a Selected Shape	185
	Sidebar: Shape-Related Object Classes	188
	Use an AutoShape to Create a Logo.	189
	Use Grouped Shapes to Create Macro Buttons.	196
	Sidebar: Selecting Multiple Items	200
	Exploring Chart Objects	201
	Create a Chart.	201
	Sidebar: The Current Selection and Charts	203
	Synchronize Two Charts	203
	Format the Plot Area of a Chart.	206
	Key Points	207
7	Control Visual Basic	209
	Using Conditionals.	210
	Make a Decision	210
	Make a Double Decision	212
	Ask Yourself a Question	214
	Test for a Valid Entry	215
	Ask with a Message	217
	Creating Loops	220
	Loop Through a Collection by Using a For Each Loop.	220
	Loop with a Counter by Using a For Loop.	222
	Loop Indefinitely by Using a Do Loop	225
	Managing Large Loops	228
	Set a Breakpoint.	229
	Set a Temporary Breakpoint.	232
	Show Progress in a Loop.	233
	Key Points	235

8	Extend Excel and Visual Basic	237
	Creating Custom Functions	238
	Use a Custom Function from a Worksheet	239
	Add Arguments to a Custom Function.	240
	Make a Function Volatile.	242
	Make Arguments Optional	243
	Use a Custom Function from a Macro	244
	Handling Errors	245
	Syntax Errors	245
	Compiler Errors.	246
	Logic Errors	246
	Run-Time Errors	246
	Ignore an Error	247
	Ignore an Error Safely by Using a Subroutine	249
	Add Arguments to Generalize a Subroutine	251
	Check for an Error	252
	Loop Until an Error Goes Away	254
	Trap an Error	255
	Key Points	259
9	Launch Macros with Events	261
	Creating Custom Command Buttons	262
	Try the ZoomIn and ZoomOut Macros.	262
	Enable the Developer Tab in the Ribbon	264
	Create a Custom Command Button	264
	Link a Command Button to a Macro.	267
	Sidebar: ActiveX Controls and Forms Controls	268
	Create an Event Handler on Your Own.	269
	Make a Button Respond to Mouse Movements	270
	Explore the Visual Basic Project.	272
	Handling Worksheet and Workbook Events	274
	Run a Procedure When the Selection Changes.	274
	Handle an Event on Any Worksheet	276

	Suppress a Workbook Event	277
	Cancel an Event	279
	Sidebar: The Ribbon and Visual Basic for Applications	281
	Key Points	281
10	Use Dialog Box Controls on a Worksheet	283
	Using a Loan Payment Calculator	284
	Create a Loan Payment Model	284
	Use the Loan Payment Model	286
	Creating an Error-Resistant Loan Payment Calculator	287
	Restrict the Years to a Valid Range	288
	Restrict the Down Payment to Valid Values	289
	Restrict the Interest Rate to Valid Values	291
	Retrieving a Value from a List	292
	Prepare a List of Cars	293
	Retrieve the Price from the List	295
	Set the Column Widths	296
	Protecting the Worksheet	297
	Create an Event Handler for the Combo Box	297
	Protect the Worksheet	299
	Key Points	301
11	Create a Custom Form	303
	Creating a Form's User Interface	304
	Create the Form	305
	Add Option Buttons	306
	Add a Check Box with a Related Text Box	309
	Initialize the Text Box	311
	Add Command Buttons	314
	Set the Tab Order for Controls	316
	Preparing a Form's Functionality	317
	Create Custom Views on a Worksheet	317
	Create a Macro to Switch Views	320
	Dynamically Hide Columns	321

Implementing a Form	325
Implement Option Buttons	325
Implement a Check Box	327
Check for Errors in an Edit Box	328
Print the Report	329
Launch the Form	331
Key Points	333
Appendix A Complete Enterprise Information System	335
Index	339



What do you think of this book? We want to hear from you!

Microsoft is interested in hearing your feedback so we can continually improve our books and learning resources for you. To participate in a brief online survey, please visit:

www.microsoft.com/learning/booksurvey/

About the Author

Reed Jacobson is a Senior Architect with Hitachi Consulting, an international management and technology consulting firm. He worked as a Software Application Specialist for Hewlett-Packard for 10 years and ran his own consulting firm for 5 years.

Reed received a BA degree in Japanese and Linguistics. He also received an MBA degree from Brigham Young University and a graduate fellowship to study Linguistics at Cornell University.

In addition to authoring this book, Reed is the author of *Excel Trade Secrets for Windows*, *Microsoft Excel Advanced Topics Step by Step*, *Microsoft Office 2000 Expert Companion*, and *Microsoft SQL Server Analysis Services Step by Step*. He has given presentations at Microsoft and other conferences, and has taught courses around the world.




Features and Conventions of This Book

This book has been designed to lead you step by step through all the tasks you are most likely to want to perform when creating macros in Microsoft Office Excel 2007. If you start at the beginning and work your way through all the exercises, you will gain enough proficiency to be able to perform many types of tasks by using macros. Each topic is self contained, but later chapters do assume that you know the information presented in earlier chapters. If you have worked with a previous version of Excel, or if you completed all the exercises and later need help remembering how to perform a procedure, the following features of this book will help you look up specific tasks related to Excel 2007 macros:

- **Detailed table of contents.** Look up the topic you want in the list of the topics and sidebars within each chapter.
- **Chapter thumb tabs.** Easily locate the beginning of the chapter you want.
- **Topic-specific running heads.** Within a chapter, quickly locate the topic you want by looking at the running head of odd-numbered pages.
- **Detailed index.** Look up specific tasks and features and general concepts in the index, which has been carefully crafted with the reader in mind.
- **Companion CD.** Find the practice files needed for the step-by-step exercises, as well as a fully searchable electronic version of this book and other useful resources.

If you are new to Excel 2007, you might not have had much time to explore the Microsoft Office Fluent user interface, which was introduced with the 2007 Microsoft Office System. The step-by-step instructions in this book often tell you to click buttons on the Office Fluent Ribbon, identifying the tab to click and the group in which the button is located. You should have no difficulty following these instructions.

You can save time when you use this book by understanding how the *Step by Step* series shows special instructions, keys to press, buttons to click, and so on. The table on the next page tells you what you need to know.

Convention	Meaning
	This icon indicates a reference to the book's companion CD.
USE	This paragraph before the first exercise in a chapter indicates the practice files that you will use when working through the exercises.
BE SURE TO	This paragraph before the first exercise in a chapter indicates any pre-requisite requirements that you should attend to before beginning the exercise, or actions you should take to restore your system after completing the exercise.
OPEN	This paragraph before the first exercise in a chapter indicates files that you should open before beginning the exercise.
CLOSE	This paragraph at the end of a chapter provides instructions for closing open files or programs before moving on to another topic.
1 2	Blue numbered steps guide you through step-by-step exercises.
1 2	Black numbered steps guide you through procedures in expository text.
→	An arrow indicates an exercise that has only one step.
See Also	These paragraphs direct you to more information about a given topic in this book or elsewhere.
Troubleshooting	These paragraphs provide a helpful hint or information about other available options.
Tip	These paragraphs provide a helpful hint or shortcut that makes working through a task easier, or information about other available options.
Important	These paragraphs point out information that you need to know to complete a procedure.
Ctrl+Home	A plus sign (+) between two key names means that you must hold down the first key while you press the second key. For example, "press Ctrl+Home" means "hold down the Ctrl key while you press the Home key."
Program interface elements	In steps, the names of program elements, such as buttons, commands, and dialog boxes, are shown in black bold characters.
User input	Anything you should type appears in blue bold characters.
<i>Italic</i>	Italic font is used for emphasis and to introduce new terms.

Using the Book's CD

The companion CD included with this book contains the practice files you'll use as you work through the book's exercises, as well as other electronic resources that will help you learn how to use VBA macros with Microsoft Office Excel 2007.

What's on the CD?

The following table lists the practice files supplied on the book's CD. Note that some practice files are used in more than one chapter.

Tip The *ExcelVBA07SBS* folder contains a subfolder named *Finished*. This folder contains the finished version of each chapter's workbook. The *Finished* folder is never explicitly referred to in the text, but it is there for your reference. If you have trouble getting a macro to work properly, you can look at the macros in the *Finished* folder to help troubleshoot the problem.

Chapter	In the <i>ExcelVBA07SBS</i> folder	In the <i>Finished</i> folder
Chapter 1: Make a Macro Do Simple Tasks	<i>Budget.xlsx</i>	<i>Chapter01.xlsm</i>
Chapter 2: Make a Macro Do Complex Tasks	<i>Nov2007.txt</i> <i>Orders.xlsx</i>	<i>Chapter02.xlsm</i>
Chapter 3: Explore Workbooks and Worksheets	None	None
Chapter 4: Explore Range Objects	<i>Ranges.xlsx</i>	<i>Chapter04.xlsm</i>
Chapter 5: Explore Data Objects	<i>Orders.xlsx</i> <i>Orders.accdb</i>	<i>Chapter05.xlsm</i>
Chapter 6: Explore Graphical Objects	<i>Graphics.xlsx</i> <i>MakeLogo.txt</i> <i>MakeMap.txt</i>	<i>Chapter06.xlsm</i>
Chapter 7: Control Visual Basic	<i>Flow.xlsx</i> <i>Flow.txt</i>	<i>Chapter07.xlsm</i>

Chapter	In the <i>Excel/VBA07SBS</i> folder	In the <i>Finished</i> folder
Chapter 8: Extend Excel and Visual Basic	<i>Structure.txt</i>	<i>Chapter08.xlsm</i>
Chapter 9: Launch Macros with Events	<i>Events.txt</i>	<i>Chapter09.xlsm</i>
Chapter 10: Use Dialog Box Controls on a Worksheet	<i>Loan.xlsx</i>	<i>Chapter10.xlsm</i>
Chapter 11: Create a Custom Form	<i>Budget.xlsx</i>	<i>Chapter11.xlsm</i>
Appendix: A Complete Enterprise Information System	<i>EIS.xlsm</i> <i>Orders.accdb</i>	None

Important The companion CD for this book does not contain the Microsoft Office Excel 2007 software. You should purchase and install that program before using this book.

Minimum System Requirements

Step-by-Step Exercises

In addition to the hardware, software, and connections required to run the 2007 Microsoft Office system, you will need the following to successfully complete the exercises in this book:

- Excel 2007
- 10 MB of available hard disk space for the practice files

2007 Microsoft Office System

For this book, you will not need the complete 2007 Microsoft Office system. You will need only Excel. The following is a reference for your convenience.

Tip If you are a Microsoft .NET Developer and want to build an application based on Excel, the contents of Chapters 3, 4, 5, and 6 will be particularly useful to help you understand the Excel object model. To use Microsoft .NET to develop applications for Excel, you will need Microsoft Visual Studio 2005 with a .NET language, as well as Microsoft Visual Studio 2005 Tools for the 2007 Microsoft Office System (VSTO 2005 SE), which is downloadable from the Microsoft.com Web site.

The 2007 Microsoft Office system includes the following programs:

- Microsoft Office Access 2007
- Microsoft Office Communicator 2007
- Microsoft Office Excel 2007
- Microsoft Office Groove 2007
- Microsoft Office InfoPath 2007
- Microsoft Office OneNote 2007
- Microsoft Office Outlook 2007
- Microsoft Office Outlook 2007 with Business Contact Manager
- Microsoft Office PowerPoint 2007
- Microsoft Office Publisher 2007
- Microsoft Office Word 2007

No single edition of the 2007 Microsoft Office system installs all of the above programs. Specialty programs available separately include Microsoft Office Project 2007, Microsoft Office SharePoint Designer 2007, and Microsoft Office Visio 2007.

To run these programs, your computer needs to meet the following minimum requirements:

- 500 megahertz (MHz) processor
- 256 megabytes (MB) RAM
- CD or DVD drive
- 2 gigabyte (GB) hard disk space for installation (a portion of this disk space will be freed if you select the option to delete the installation files)

Tip Hard disk requirements will vary depending on configuration; custom installation choices may require more or less hard disk space.

- Monitor with minimum 1024 × 768 screen resolution
- Keyboard and mouse or compatible pointing device
- Internet connection, 128 kilobits per second (Kbps) or greater, for download and activation of products, accessing Microsoft Office Online and online Help topics, and any other Internet-dependent processes
- Windows Vista or later, Microsoft Windows XP with Service Pack 2 (SP2) or later, or Microsoft Windows Server 2003 or later
- Windows Internet Explorer 7 or Microsoft Internet Explorer 6 with service packs

The 2007 Microsoft Office suites, including Office Basic 2007, Office Home & Student 2007, Office Standard 2007, Office Small Business 2007, Office Professional 2007, Office Ultimate 2007, Office Professional Plus 2007, and Office Enterprise 2007, all have similar requirements.

Installing the Practice Files

You need to install the practice files to a suitable location on your hard disk before you can use them in the exercises. Follow the steps below.

Note If for any reason you are unable to install the practice files from the CD, the files can also be downloaded from the Web at <http://www.microsoftpressstore.com/title/9780735624023>.

1. Remove the companion CD from the envelope at the back of the book, and insert it into the CD drive of your computer.

The Step By Step Companion CD License Terms appear. Follow the on-screen directions. To use the practice files, you must accept the terms of the license agreement. After you accept the license agreement, a menu screen appears.

Important If the menu screen does not appear, click the Start button and then click Computer. Display the Folders list in the Navigation Pane, click the icon for your CD drive, and then in the right pane, double-click the StartCD executable file.

2. Click **Practice Files**.

Important On a computer running Windows Vista, the default installation location of the practice files is *Documents\MSP\ExcelVBA07SBS*. On a computer running Windows XP, the default installation location is *My Documents\MSP\ExcelVBA07SBS*. If your computer is running Windows XP, whenever an exercise tells you to navigate to your *Documents* folder, you should instead go to your *My Documents* folder.


3. Click **Next** on the first screen, and then click **Next** to accept the terms of the license agreement on the next screen.
4. If you want to install the practice files to a location other than the default folder (*Documents\MSP\ExcelVBA07SBS*), click the **Change** button, select the new drive and path, and then click **OK**.

Important If you install the practice files to a location other than the default, you will need to substitute that path within the exercises.

5. Click **Next** on the **Custom Setup** page, and then click **Install** on the **Ready to Install the Program** screen to install the selected practice files.
6. After the practice files have been installed, click **Finish**.
7. Close the **Step by Step Companion CD** window, remove the companion CD from the CD drive, and return it to the envelope at the back of the book.

Using the Practice Files

When you install the practice files from the companion CD that accompanies this book, the files are stored on your hard disk in *Documents\MSP\ExcelVBA07SBS*. Each exercise is preceded by a paragraph that lists the files needed for that exercise and explains any preparations needed before you start working through the exercise. Here are examples:

-  **USE** the *Budget.xlsx* workbook and the *Nov2007.txt* text file. These practice files are located in the *Documents\MSP\ExcelVBA07SBS* folder.
- BE SURE TO** save *Budget.xlsx* as a macro-enabled workbook called *Chapter04* in the trusted folder that you created in Chapter 1.
- OPEN** the *Chapter04.xlsm* workbook.

You can browse to the practice files in Windows Explorer by following these steps:



Start

1. On the Windows taskbar, click the **Start** button, and then click **Documents**.
2. In your **Documents** folder, double-click **MSP**, and then double-click **ExcelVBA07SBS**.

You can browse to the practice files from an Excel 2007 dialog box by following these steps:

1. In the **Favorite Links** pane in the dialog box, click **Documents**.
2. In your **Documents** folder, double-click **MSP**, and then double-click **ExcelVBA07SBS**.

Removing and Uninstalling the Practice Files

You can free up hard disk space by uninstalling the practice files that were installed from the companion CD. The uninstall process deletes any files that you created in the *Documents\MSP\Excel\VBA07SBS* folder while working through the exercises. Follow these steps:



Start

1. On the Windows taskbar, click the **Start** button, and then click **Control Panel**.
2. In **Control Panel**, under **Programs**, click the **Uninstall a program** task.
3. In the **Programs and Features** window, click **Microsoft Office Excel VBA 2007 Step by Step**, and then on the toolbar at the top of the window, click the **Uninstall** button.
4. If the **Programs and Features** message box asking you to confirm the deletion appears, click **Yes**.

Important Microsoft Product Support Services does not provide support for this book or its companion CD.

Getting Help

Every effort has been made to ensure the accuracy of this book and the contents of its companion CD. If you do run into problems, please contact the sources listed below for assistance.

Errata & Book Support

If you find an error, please report it on our Microsoft Press site.

1. Go to www.microsoftpressstore.com.
2. In the **Search** box, enter the book's ISBN or title.
3. Select your book from the search results.
4. On your book's catalog page, find the Errata & Updates tab.
5. Click **View/Submit Errata**.

You'll find additional information and services for your book on its catalog page. If you need additional support, please e-mail Microsoft Press Book Support at mspin-put@microsoft.com. If for any reason you are unable to install the practice files from the CD, the files can also be downloaded from the web here: <http://www.microsoft-pressstore.com/title/9780735623057>.

Please note that product support for Microsoft software is not offered through the addresses above.

More Information

If your question is about Microsoft Office Excel 2007 or another Microsoft software product, and you cannot find the answer in the product's Help, please search the appropriate product solution center or the Microsoft Knowledge Base at:

support.microsoft.com

In the United States, Microsoft software product support issues not covered by the Microsoft Knowledge Base are addressed by Microsoft Product Support Services. Location-specific software support options are available from:

support.microsoft.com/gp/selfoverview/

4 Explore Range Objects

In this chapter, you will learn to:

- ✓ Use several properties to refer to Range objects from macro statements.
 - ✓ Put values and formulas into cells.
 - ✓ Simplify macros that record selections.
 - ✓ Apply formatting to ranges.
 - ✓ Use the Object Browser to learn about objects, properties, and methods.
-

The world would be much simpler if people were all the same size. You wouldn't need adjustable seats in your car; your head would never get bumped on a door frame; your feet would never dangle from a chair. Of course, you'd have new problems as well: When you went to exchange that hideous outfit you got for your birthday, you wouldn't be able to claim it was the wrong size.

When using Microsoft Visual Basic for Applications (VBA) to write macros for Microsoft Office Excel, you don't need to worry about Range objects as long as all your worksheets and data files are the same size. For example, if you never insert new rows into a budget, if you always put yearly totals in column M, and if every month's transaction file has exactly 12 columns and 120 rows, you can skip this chapter because the macro recorder can take care of dealing with ranges for you.

But in the real-live human world, people are different sizes, and consequently clothes come in different sizes and cars have adjustable seats. And in the real-live worksheet world, models and data files have different—and changing—sizes, and your macros need to fit them. Excel provides many methods and properties for working with Range objects. In this chapter, you'll explore Range objects and in the process learn how you can use the Object Browser to learn about any new, unfamiliar object.



Important Before you complete this chapter, you need to install the practice files from the book's companion CD to their default locations. See "Using the Book's CD" on page xv for more information.



USE the *Ranges.xlsx* workbook. This practice file is located in the *Documents\MSP\ExcelVBA07SBS* folder.

BE SURE TO save a macro-enabled copy of the *Ranges.xlsx* workbook as *Chapter4.xlsm* in the trusted folder location you created in Chapter 1.

OPEN the *Chapter4.xlsm* workbook.

Referring to a Range

A macro that needs to work with ranges of differing sizes must be flexible. In this section, you'll learn various ways to refer to a range. The examples in this section don't *do* anything except reference ranges within a list, but these are all techniques you'll use many times as you work with ranges. Later in the chapter, you'll use these techniques in more practical contexts.

Refer to a Range by Using an Address

The Range property is a useful and flexible way of retrieving a reference to a range. The Range property allows you to specify the address of the range you want. You can use the Object Browser to see how to use the Range property.

1. In the *Chapter04* workbook, right-click a worksheet tab, and then click **View Code** on the shortcut menu to display the Visual Basic editor.

Rearrange the Excel and Visual Basic editor windows so that you can see them side by side.

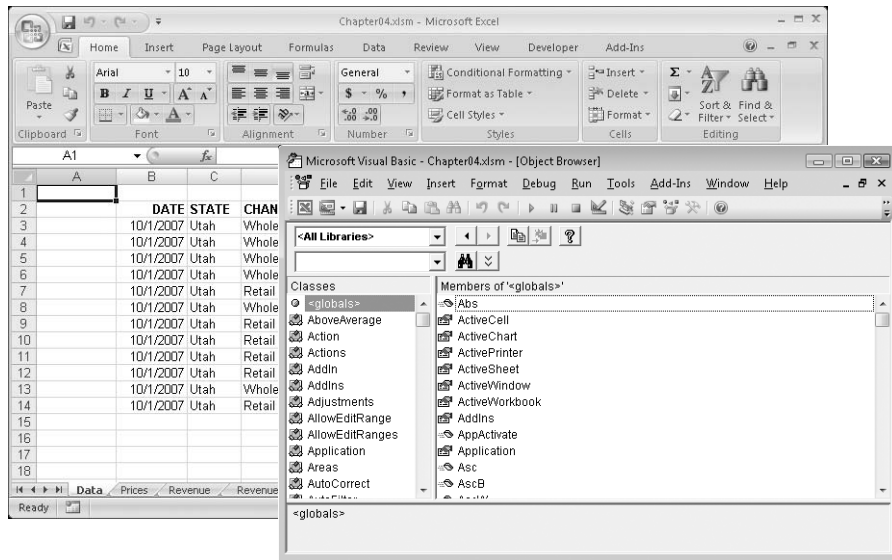
2. In the Visual Basic editor, click the **Object Browser** toolbar button.

See Also If you want to change the Object Browser into a dockable window, see the sidebar titled "Dockable Views" in Chapter 3, "Explore Workbooks and Worksheets."



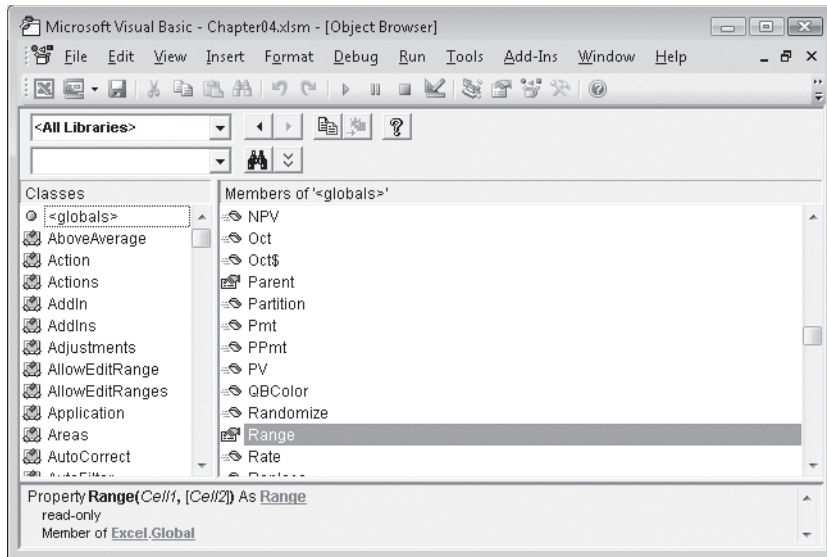
Object Browser

The Object Browser appears in the space normally held by the code window. In essence, the Object Browser consists of two lists. The list on the left is a list of object class names. The list on the right is a list of members—methods and properties—available for the currently selected object class. At the top of the list of classes is a special object class named *<globals>*. The *<globals>* object is not a real object class, but it includes in its list of members all the methods and properties you can use without specifying an object. These are the methods and properties you use to start a statement.



3. In the **Classes** list, select the **<globals>** object, click in the **Members of '<globals>'** list, and press the R key to scroll to the first member that begins with the letter R. Then select the **Range** property.

The box at the bottom of the Object Browser displays information about the Range property. This property takes two arguments. The brackets around the second argument indicate that it is optional. The Range property returns a reference to a Range object.



4. Right-click the **Range** property name in the **Members** list, and click **Copy** on the shortcut menu.
 5. Click the **View** menu, and click **Immediate Window**.
 6. Right-click the **Immediate** window, and click **Paste**.
This is equivalent to using the Complete Word command to enter the function name.
 7. After the **Range** property, type an opening parenthesis (Visual Basic will display the argument list), and then type **"B2"** followed by a closing parenthesis and a period. Then type **Select**.
- The complete statement is `Range("B2").Select`. You need the quotation marks around the range definition because this is the *name* of the range, not the item number of a member of a collection.
8. Press Enter to select cell **B2** on the active worksheet.
 9. Type `Range("B2:H2").Select` and press Enter.

The first argument of the range property can be a multicell range. In fact, it can be anything that Excel recognizes.

	A	B	C	D	E	F	G	H	
1									
2									
3			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
4		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
5		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
6		10/1/2007							
7		10/1/2007							

Immediate

Range ("B2:H2") .Select

- 10.** Type `Range("H14").Select` and press Enter to select the lower-right corner of the list of values. Then type `Range(Selection, "B2").Select` and press Enter.

This selects the range from cell H14 (the current selection) to cell B2 (the upper left cell of the list). The arguments to the Range property do not have to be strings; they can also be references to range objects. A common use of the two-argument form of the Range property is to select the range that extends from the currently selected range to some fixed point at the top of the worksheet.

- 11.** Type `?Selection.Count` and press Enter.

The number `91` appears in the Immediate window. There are 91 cells in the currently selected range. If you don't specify otherwise, Excel treats a range object as a collection of cells. If you want to know the number of rows or columns in the range, you can do that by using specialized properties, as you will learn in the section titled "Refer to a Range as a Collection of Rows or Columns," later in this chapter.

Tip As you learned in Chapter 3, typing a question mark before an expression in the Immediate window allows you to display the value of that expression.

The Range property is a flexible way of establishing a link to an arbitrary Range object. You can use either a single text string that contains any valid reference as an argument to the Range property or two arguments that define the end points of a rectangular range. Once you have the resulting reference to a range object, you can use any of the members that appear in Object Browser for the Range class.

Refer to a Range as a Collection of Cells

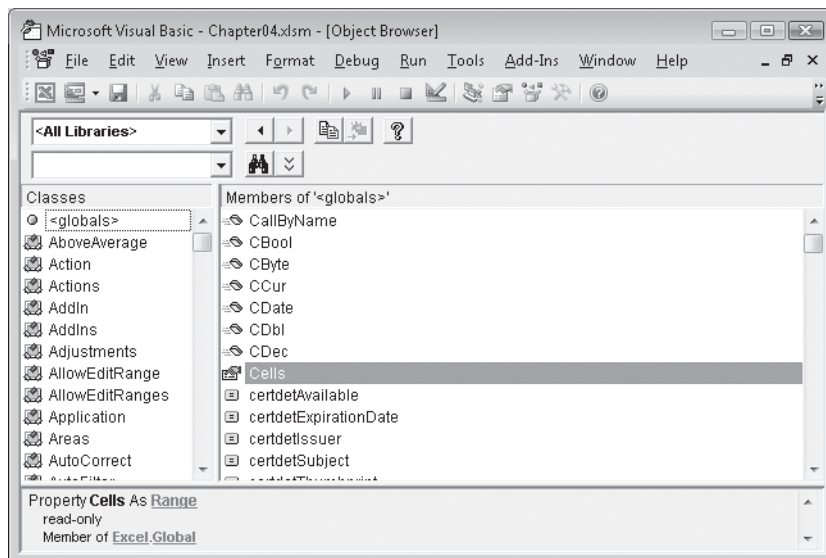
Multiple worksheets can exist in a workbook, and the Worksheets collection is defined as an object class. A Worksheets object has a list of methods and properties that is separate from a Worksheet object.

Similarly, multiple cells exist on a worksheet. You might expect that Excel would have a Cells collection object. But a collection of cells is more complicated than a collection of worksheets because cells come in two dimensions—rows and columns. For example, you can think of the range A1:B3 as a collection of six cells, as a collection of three rows, or as a collection of two columns.

Excel therefore has three properties that look at a range as a collection. The first of these—the Cells property—returns a collection of cells. However, this is not a separate class. The result of the Cells property is still a Range object, and it can use any of the methods or properties of any other Range object. Because Excel thinks of any range, by default, as a collection of cells, you typically use the Cells property as an alternative to the Range property—using numbers, rather than text strings.

1. In the **Object Browser**, with the **<globals>** object selected in the list of classes, select the **Cells** property from the list of members.

The description at the bottom of the Object Browser indicates that the Cells property returns a Range object.



2. In the **Immediate** window, type **Cells.Select** and press Enter.

This selects all the cells on the worksheet. This is equivalent to clicking the box at the upper left corner of the worksheet, between the column A heading and the row 1 heading.

	A	B	C	D	E	F	G	H	I
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah						
5		10/1/2007	Utah						

Immediate

```
Cells.Select
```

3. Type `Cells.Item(5).Select` and press Enter.

This selects cell E1, the fifth cell in the first row. The Cells property returns the range of all the cells on the worksheet as a collection. An individual item in the collection is a cell.

	A	B	C	D	E	F	G	H	
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007							

Immediate

```
Cells.Item(5).Select
```

4. Type `Cells.Item(16383).Select` and press Enter.

This selects cell XFC1, the next to the last cell in the first row. Excel 2007 now allows 16384 cells in a single row.

	XEY	XEZ	XFA	XFB	XFC	XFD
1						
2						
3						
4						

Immediate

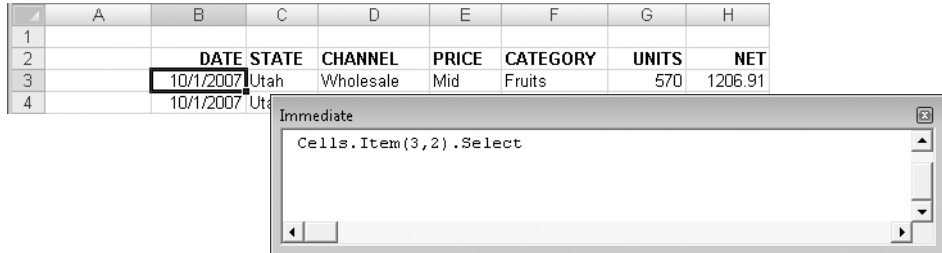
```
Cells.Item(16383).Select
```

5. Type `Cells.Item(16385).Select` and press Enter.

This selects cell A2, the first cell in the second row. When you use a single number to select an item in the Cells collection, the number wraps at the end of each row. Since each row of the worksheet contains 16384 cells, cell 16385 is the first cell on the second row.

6. Type `Cells.Item(3,2).Select` and press Enter.

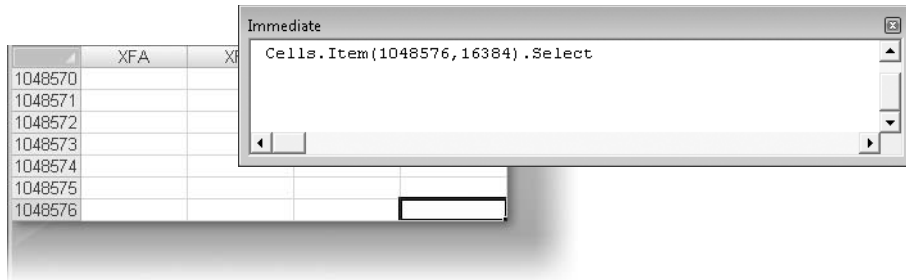
This selects cell B3, the third row and second column in the worksheet. Unlike most other collections, the Cells collection allows you to specify an item by using both the row and column values.



Important In previous versions of Excel, the expression `Cells.Item(257)` referred to cell A2. In Excel 2007, it now refers to cell IW1, the 257th cell in the first row. In order to write macros that work in multiple versions, you should always use the row and column specification in the Cells function. Another consequence of the larger size of the worksheet is that you cannot use the expression `Cells.Count` to retrieve the number of cells on the worksheet, because the number is too big. This is unlikely to ever be a problem, but it illustrates the expanded size of the worksheet grid.

7. Type `Cells.Item(1048576,16384).Select` and press Enter.

This selects cell XFD1048576, the bottom right cell in the worksheet. In case you wonder, these bizarre-looking numbers are really simple powers of 2. You could select the same cell by using the expression `Cells.Item(2^20,2^14)`. You could also use the Range property—`Range("XFD1048576")`.



8. Type `Cells(1).Select` and press Enter to select cell A1.

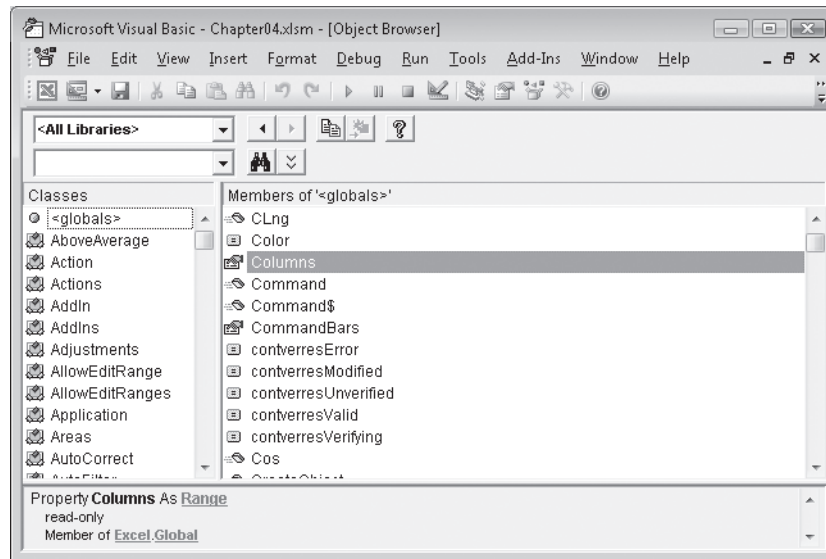
As with other collections, when you use the Cells property to get a collection of cells, you can leave out the Item method, and simply put the argument after the Cells property. The expression `Cells(1)` is equivalent to `Cells.Item(1)`, which is equivalent to `Range("A1")`. All these expressions can be used interchangeably.

Refer to a Range as a Collection of Rows or Columns

In addition to referring to the worksheet range as a collection of cells, you can also think of it as a collection of rows or as a collection of columns. Analogous to the Cells property, the Rows property returns a collection of rows and the Columns property returns a collection of columns. These properties return collections, do not have their own object classes, and return Range objects.

1. In the **Object Browser**, with the **<globals>** object selected in the list of classes, select the **Columns** property in the list of **Members**.

The description shows that this property, similar to the Range property and the Cells property, returns a Range object.



2. In the **Immediate** window, type **Columns.Select** and press Enter.

This selects all the cells on the worksheet, exactly the same as Cells.Select. The difference between the two properties appears when you use the Item property to index into a single item in the collection.

3. Type **Columns(3).Select** and press Enter.

This selects column C, the third column on the worksheet.

	A	B	C	D	E	F	G	H	
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007							
6		10/1/2007							

Immediate
 Columns(3) .Select

4. Type `Columns("D").Select` and press Enter.

This selects column D. When you specify a column by letter, you are giving the *name* of the item and must enclose it in quotation marks.

	A	B	C	D	E	F	G	H	
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007	Utah	Wholesale	Mid	Flowers	285	623.23	
6		10/1/2007							
7		10/1/2007							

Immediate
 Columns("D") .Select

5. Type `Columns("B:H").Select` and press Enter.

This selects the range of columns from B through H. The only way to specify a range of columns within the collection is by using the column letter names.

	A	B	C	D	E	F	G	H	I
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007							
6		10/1/2007							
7		10/1/2007							

Immediate
 Columns("B:H") .Select

6. Type `Rows(2).Select` and press Enter.

This selects row 2. With rows, the name of an item is also a number. The expressions `Rows(2)` and `Rows("2")` are functionally equivalent.

	A	B	C	D	E	F	G	H	I
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007							

Immediate

```
Rows(2).Select
```

7. Type `Rows("3:14").Select` and press Enter.

This selects a range of rows. The only way to specify a range of rows within the collection is by using the row numbers as a name—that is, by enclosing them in quotation marks.

	A	B	C	D	E	F	G	H	I
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007	Utah	Wholesale	Mid	Flowers	285	623.23	
6		10/1/2007	Utah	Wholesale	Mid	Herbs	285	622.3	
7		10/1/2007	Utah	Retail	Mid	Fruits	245	856.83	
8		10/1/2007	Utah	Wholesale	Mid	Tools	135	303.75	
9		10/1/2007	Utah	Retail	Mid	Herbs	65	292.5	
10		10/1/2007	Utah	Retail	Mid	Tools	56	252	
11		10/1/2007	Utah	Retail	Mid	Books	40	180	
12		10/1/2007							
13		10/1/2007							
14		10/1/2007							
15									

Immediate

```
Rows("3:14").Select
```

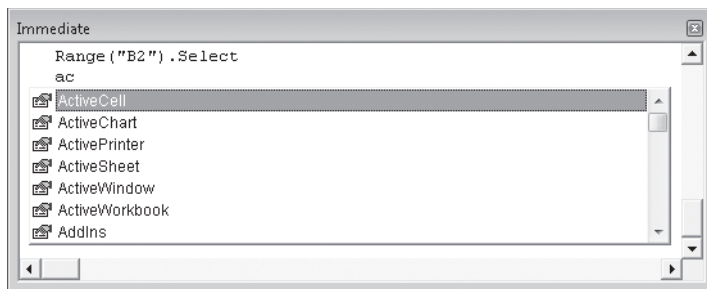
The `<globals>` object in the Object Browser includes three properties that return all the cells of a worksheet—`Cells`, `Columns`, and `Rows`. In each case, you get a reference to a `Range` object, but the properties return that object as a collection of cells, columns, or rows, respectively. There are no object classes for `Cells`, `Columns`, and `Rows`. These are simply different ways of representing the same `Range` object.

Refer to a Range Based on the Active Cell

Many times when writing a macro you want to refer to a range that is somehow related to the active cell or to the current selection. The macro recorder uses the Selection property to refer to the selected range and the ActiveCell property to refer to the one active cell. A Range object has useful properties that can extend the active cell or the selection to include particularly useful ranges.

1. In the **Immediate** window, type `Range("B2").Select` and press Enter.
This selects the upper left cell of the sample list.
2. In the **Object Browser**, with the `<globals>` object selected in the **Classes** list, select the **ActiveCell** property.
The description at the bottom of the Object Browser shows that this property returns a Range object.
3. In the **Immediate** window, click the **Edit** menu, and then click **Complete Word**. In the list of members, click **ActiveCell**.

Tip When you use the Complete Word command at the beginning of a statement—whether in a macro or in the Immediate window—the Auto List displays all the members of the `<globals>` object. If you like using the keyboard, you can press Ctrl+Space to display the list of members, type partial words and use arrow keys to select the desired member, and then press the Tab key to insert the member into the statement.



4. Type a period (.). Then type `CurrentRegion.Select` to create the statement `ActiveCell.CurrentRegion.Select`, and then press Enter.

This selects the entire sample list. The CurrentRegion property selects a rectangular range that includes the original cell and is surrounded by either blank cells or the edge of the worksheet. It is hard to overstate the usefulness of the CurrentRegion property.

	A	B	C	D	E	F	G	H	I
1									
2									
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007	Utah	Wholesale	Mid	Flowers	285	623.23	
6		10/1/2007	Utah	Wholesale	Mid	Herbs	285	622.3	
7		10/1/2007	Utah	Retail	Mid	Fruits	245	856.83	
8		10/1/2007	Utah	Wholesale	Mid	Tools	135	303.75	
9		10/1/2007	Utah	Retail	Mid	Herbs	65	292.5	
10		10/1/2007	Utah	Retail	Mid	Tools	56	252	
11		10/1/2007	Utah	Retail	Mid	Books	40	180	
12		10/1/2007	Utah	Retail	Mid	Flowers	35	157.5	
13		10/1/2007							
14		10/1/2007							
15									

Immediate

```
Range("B2").Select
ActiveCell.CurrentRegion.Select
```

5. Type `ActiveCell.EntireColumn.Select` and press Enter.

This selects all of column B because the active cell was cell B2. Because the starting range was the active cell—not the entire selection—the `EntireColumn` property returned a reference to only one column. Because the initial active cell—B2—is still within the selection, it is still the active cell.

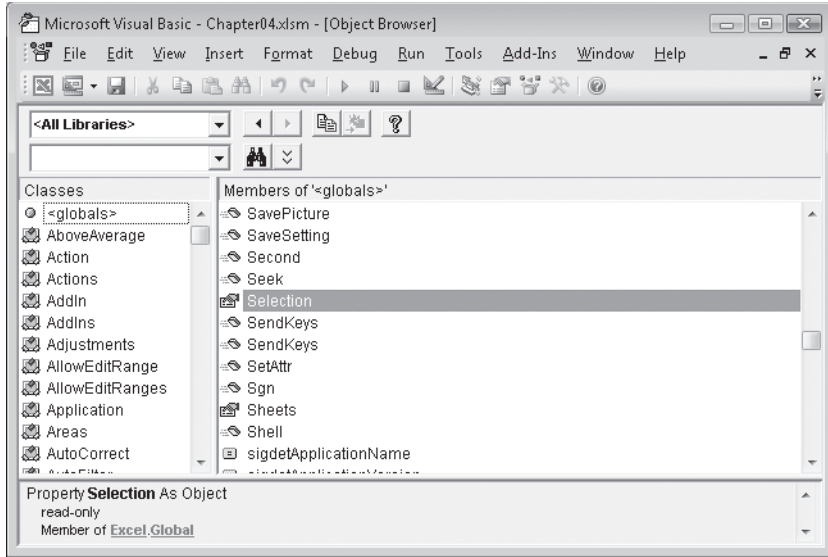
	A	B	C	D	E	F	G	H
1								
2								
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79
5		10/1/2007						
6		10/1/2007						
7		10/1/2007						

Immediate

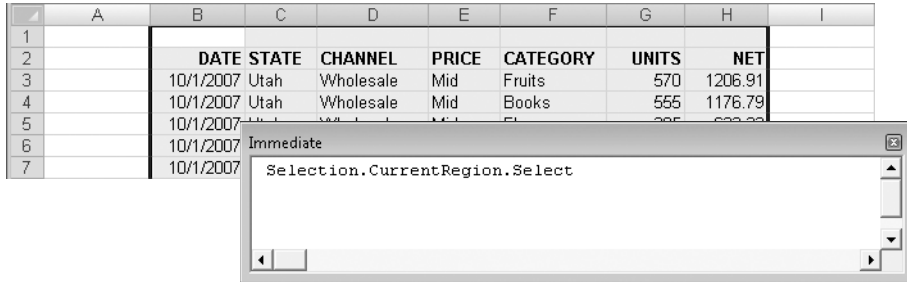
```
ActiveCell.EntireColumn.Select
```

6. In the **Object Browser**, with the `<globals>` object selected, select the **Selection** property in the list of members.

The description at the bottom indicates that the `Selection` property returns an *object*, not a `Range`. The `Selection` property returns a `Range` object only when cells are selected. If shapes or parts of a chart are selected, this global property returns a different object type. Because the `Selection` object can return a different object type at different times, it does not display an Auto List the way the `ActiveCell` property does.

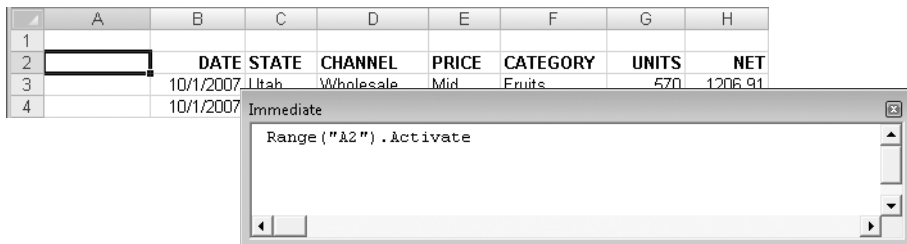


- In the Immediate window, type `Selection.CurrentRegion.Select` and press Enter. This selects the range B1:H14—the entire sample list plus the one row above it. It's acting the same as if the current selection were only cell B1. When you use the CurrentRegion property with a multicell range as the starting point, it ignores everything except the top-left cell of the range as it calculates the current region.



- Type `Range("A2").Activate` and press Enter.

Because the specified cell is outside of the current selection, the Activate method behaves the same as Select.

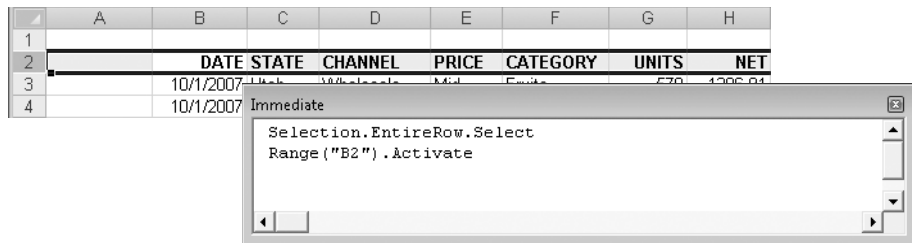


9. Type `Selection.EntireRow.Select` and press Enter.

This selects all of row 2. Because the selection is a single cell, you would get exactly the same result by using `ActiveCell.EntireRow.Select`.

10. Type `Range("B2").Activate` and press Enter.

Because the specified cell is within the selected range, this statement does not change the selection, but it does move the active cell to a new location within the range. If you activate a cell that is not within the current selection, the Activate method behaves the same as Select.



The Selection and ActiveCell properties are useful as starting points for deriving other ranges. The ActiveCell property always returns a reference to a Range object, and therefore displays a convenient Auto List when you are entering a statement. The Selection property returns a reference to a Range object only when a range is actually selected, and thus it does not display an Auto List.

Refer to Subsets of a Range

When you reference a range by using a property from the <globals> object—for example, Range, Cells, Columns, or Rows—you get a range based on the entire active worksheet. These same properties also exist as properties of a Range object. The easiest way to work with properties of a Range object is to declare a variable as a Range. Then the Auto List displays the methods and properties as you type a statement, even if you use the Selection property—which does not display Auto Lists—to assign the range to the variable.

1. In the Visual Basic editor, click **Insert**, and then click **Module**.
2. Type `Sub TestRange` and press Enter.
Visual Basic adds parentheses and an `End Sub` statement.
3. Type `Dim myRange As Range` and press F8 twice to initialize the variable.
4. In the **Immediate** window, type `Set myRange = Range("B2")` and press Enter. Then type `myRange.Select` and press Enter again.

This selects cell B2, confirming that the variable contains a reference to that cell.

- Click the **Object Browser** button. In the list of classes, select the **Range** class. Then in the list of members, select the **Range** property.

This Range property appears very similar to the Range property of the <globals> object. It behaves, however, *relative* to a starting range.

- In the **Immediate** window, type `myRange.Range("A1:G1").Select` and press Enter.

This does not select the range A1:G1. Rather, it selects the range B2:H2. If you think of cell B2 as the upper left cell of an imaginary worksheet, the range A1:G1 of that imaginary worksheet would correspond to the range B2:H2 of the real worksheet.

	A	B	C	D	E	F	G	H	
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007							
6		10/1/2007							
7		10/1/2007							

Immediate

```
Set myRange = Range("B2")
myRange.Range("A1:G1").Select
```

- Type `Set myRange = myRange.CurrentRegion` and press Enter. Then type `myRange.Select` and press Enter again.

Given that myRange already referred to cell B2, which is inside the sample list, the first statement references the entire sample list, and the second confirms that the variable contains a reference to the appropriate range.

- Type `myRange.Cells.Item(2,6).Select` and press Enter.

This selects the first data value in the Units column—row 2 and column 6 within the data region.

	A	B	C	D	E	F	G	H	I
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007							

Immediate

```
Set myRange = myRange.CurrentRegion
myRange.Select
myRange.Cells.Item(2,6).Select
```

- Type `myRange.Rows(2).Select` and press Enter.

This selects the second row of values in the list, even though they exist in row 3 of the worksheet. A single row from the collection referenced by the global Rows property includes the entire row of the worksheet; the Rows property of a Range object includes only the cells within the range.

10. Type `myRange.Rows(myRange.Rows.Count).Select` and press Enter.

This selects the last row of the list. Because the Rows property returns a collection, you can use the Count property to find the number of items in the collection. That count can then serve as an index into the same collection.

11		10/1/2007	Utah	Retail	Mid	Books	40	180
12		10/1/2007	Utah	Retail	Mid	Flowers	35	157.5
13		10/1/2007	Utah	Wholesale	Mid	Books	30	67.5
14		10/1/2007	Utah	Retail	Mid	Books	28	126
15								

Immediate

```
myRange.Rows(2).Select
myRange.Rows(myRange.Rows.Count).Select
```

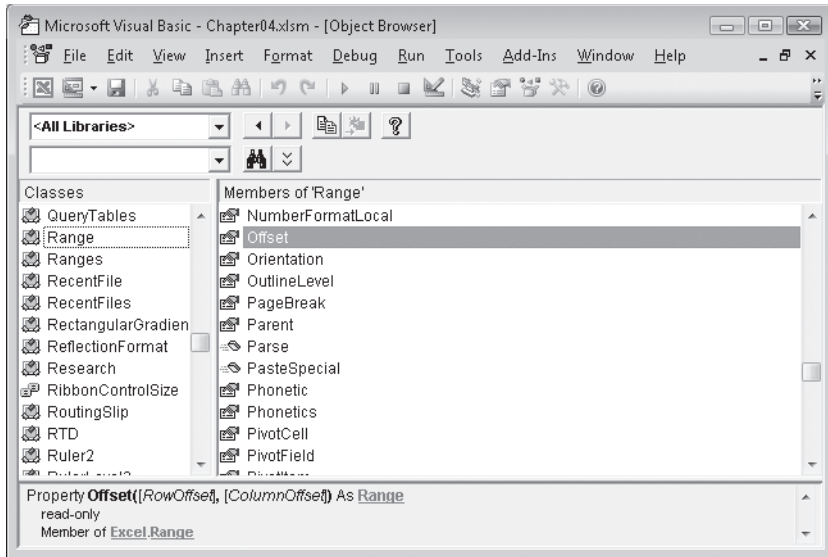
When you use the Range, Cells, Columns, or Rows properties as members of a Range object, the resulting ranges are relative to the upper-left cell of that range. Contrast this to when you use the same functions from the global group—or as members of the Application object or of a Worksheet object. With anything other than a Range object, these functions return ranges that are relative to the upper-left cell of the worksheet.

Refer to a Relative Range

Excel has other properties that can calculate a new range based on one or more existing ranges. Two of these properties do not exist in the list of global members; they exist only as members of a Range object: the Offset property references a range shifted down, up, left, or right from a starting range, and the Resize property references a range with a different number of rows or columns from a starting range. An additional property, the Intersect property, does appear in the list of global members. It is particularly valuable when you need to “trim away” part of a range, such as when you want to remove the header row from the current region.

1. In the **Object Browser**, select **Range** in the **Classes** list. Then, in the **Members** list, select the **Offset** property.

The description indicates that this property has two arguments—RowOffset and ColumnOffset, both of which are optional—and that it returns a Range object.



2. In the Immediate window, type `myRange.Offset(1).Select` and press Enter.

This selects a range identical in size and shape to the range stored in the variable, but shifted down by one cell. The first argument to the Offset property indicates the number of rows down to shift the range; the second argument indicates how many columns to the right to shift the range. Omitting an argument is the same as using zero and does not shift the range in that direction.

	A	B	C	D	E	F	G	H	I
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									
16									

DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91
10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79
10/1/2007	Utah	Wholesale	Mid	Flowers	285	623.23
10/1/2007	Utah	Wholesale	Mid	Herbs	285	622.3
10/1/2007	Utah	Retail	Mid	Fruits	245	856.83
10/1/2007	Utah	Wholesale	Mid	Tools	135	303.75
10/1/2007	Utah	Retail	Mid	Herbs	65	292.5
10/1/2007	Utah	Retail	Mid	Tools	56	252
10/1/2007	Utah	Retail	Mid	Books	40	180
10/1/2007	Utah	Retail	Mid	Flowers	35	157.5
10/1/2007	Utah	Wholesale	Mid	Books	30	67.5
10/1/2007						

Immediate

```
myRange.Offset(1).Select
```

Tip To understand the Offset property, think of yourself as standing on the upper-left cell of the initial range. Face the bottom of the worksheet, and step forward the number of steps specified in the first argument. Zero steps means no movement. Negative steps are backwards. Then face the right side of the worksheet and do the same with the number of steps specified in the second argument. The resulting range is the same size and shape as the original one, but it begins on the cell you end up standing on.

3. In the **Object Browser**, select **Range** in the **Classes** list. Then, in the list of members, select the **Resize** property.

The description indicates that this property has two arguments—**RowSize** and **ColumnSize**, both of which are optional—and that it returns a Range object.

4. In the **Immediate** window, type `myRange.Offset(1).Resize(5).Select` and press Enter.

This selects the first five rows of data. The Offset property shifts the range down to omit the heading row. The Resize function changes the size of the resulting range. The first argument to the Resize property is the number of rows for the result range; the second is the number of columns for the result range. Omitting an argument is the same as keeping the size of the original range for that direction.

	A	B	C	D	E	F	G	H	I
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	
5		10/1/2007	Utah	Wholesale	Mid	Flowers	285	623.23	
6		10/1/2007	Utah	Wholesale	Mid	Herbs	285	623.23	
7		10/1/2007	Utah	Wholesale	Mid	Herbs	285	623.23	
8		10/1/2007	Utah	Wholesale	Mid	Herbs	285	623.23	

Immediate

```
myRange.Offset(1).Resize(5).Select
```

5. Type `myRange.Offset(1,5).Resize(1,2).Select` and press Enter.

This selects the range G3:H3, which happens to be the numeric values in the first row of the body of the list.

	A	B	C	D	E	F	G	H	I
1									
2			DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
3		10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91	
4		10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79	

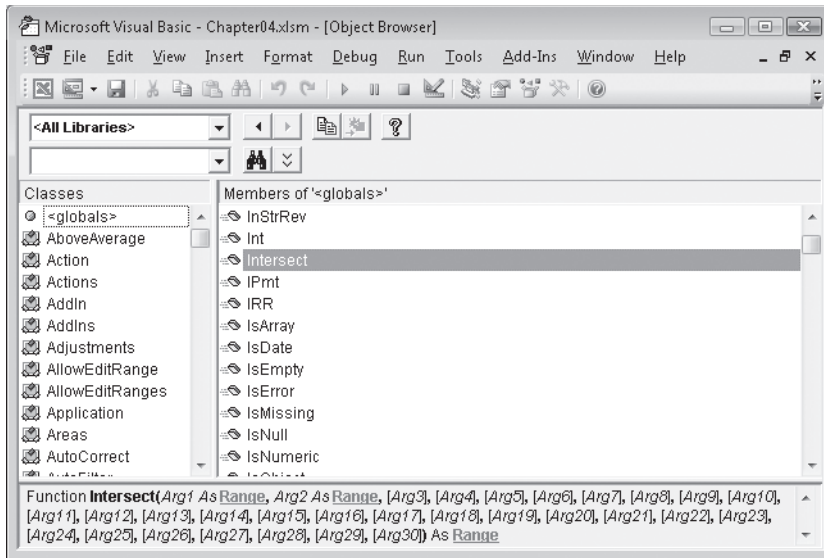
Immediate

```
myRange.Offset(1,5).Resize(1,2).Select
```

Tip The combined functionality of the Offset and Resize properties is equivalent to that of the OFFSET function available on worksheets.

- In the **Object Browser**, with the `<globals>` object selected in the list of classes, select the **Intersect** method in the **Members** list.

The description shows that this method returns a Range object, but it also shows that it can take up to 30 arguments! In practice, you usually use two arguments, and you can see that the first two arguments are required. The Object Browser shows that the first two arguments must be range objects, but if you use more than two arguments, they do all need to be ranges. You can use the Intersect method in conjunction with the Offset method to remove headings from the current region.



- In the **Immediate** window, type `Intersect(myRange, myRange.Offset(1)).Select` and press Enter.

This selects the range B3:H14, which is the entire list except the heading row. You often need to manipulate the body of a list separately from the heading. By using a range as the first argument of the Intersect method, and then an offset version of the range as the second argument, you can trim off portions of the range.

	A	B	C	D	E	F	G	H	I
1									
2									
3									
4									
5									
6									
7									
8									
9									
10									
11									
12									
13									
14									
15									

DATE	STATE	CHANNEL	PRICE	CATEGORY	UNITS	NET
10/1/2007	Utah	Wholesale	Mid	Fruits	570	1206.91
10/1/2007	Utah	Wholesale	Mid	Books	555	1176.79
10/1/2007	Utah	Wholesale	Mid	Flowers	285	623.23
10/1/2007	Utah	Wholesale	Mid	Herbs	285	622.3
10/1/2007	Utah	Retail	Mid	Fruits	245	856.83
10/1/2007	Utah	Wholesale	Mid	Tools	135	303.75
10/1/2007	Utah	Retail	Mid	Herbs	65	292.5
10/1/2007	Utah	Retail	Mid	Tools	56	252
10/1/2007	Utah	Retail	Mid	Books	40	180
10/1/2007	Utah	Retail	Mid	Flowers	35	157.5
10/1/2007	Utah	Wholesale	Mid	Books	30	67.5

Immediate

```
Intersect(myRange, myRange.Offset(1)).Select
```

8. Press F5 to end the macro.

The Offset and Resize properties, along with the EntireRow, EntireColumn, and CurrentRegion properties and the Intersect method, provide you with flexible tools for calculating new Range objects based on an original starting range. Often, the easiest way to work within a range is to first use the CurrentRegion property to establish the base range, and then use the Offset property and the Intersect method to manipulate the range.

Enhancing Recorded Selections

When you record a macro, the macro recorder dutifully follows all your actions, including selecting ranges before acting on them. You can make a macro do less work—and make it easier to read—by eliminating unnecessary selection changes. A powerful technique for eliminating unnecessary changes to the selection begins with watching for a statement ending in Select followed by one or more statements beginning with Selection or ActiveCell. What you do next depends on whether a single Selection (or ActiveCell) statement follows the Select statement or whether a group of statements follows.

Simplify Select...Selection Pairs

When a single Selection statement follows a Select statement, you can collapse the two statements into one. Record and simplify a macro that puts the names of the months across the top of a worksheet.

1. In Excel, insert a blank worksheet and start recording a macro named **LabelMonths**. Type the labels **January**, **February**, and **March** in the cells **B1**, **C1**, and **D1**.

	A	B	C	D	E
1		January	February	March	
2					
3					

2. Turn off the recorder, and then edit the macro.

The macro should look similar to the following code. (Your macro might be slightly different, depending on the key you press to enter the values into the cells.)

```
Sub LabelMonths()
    Range("B1").Select
    ActiveCell.FormulaR1C1 = "January"
    Range("C1").Select
    ActiveCell.FormulaR1C1 = "February"
    Range("D1").Select
    ActiveCell.FormulaR1C1 = "March"
    Range("D2").Select
End Sub
```

For each cell, the word *Select* appears at the end of one statement followed by either the word *Selection* or *ActiveCell* at the beginning of the next statement. You can delete both words, leaving only a single period. If a *Select* statement is the last one in a macro, you can delete it entirely.

3. Remove the unnecessary selections from the **LabelMonths** macro by deleting **Select** and **ActiveCell** each time they appear.

The final macro should look like this:

```
Sub LabelMonths()
    Range("B1").FormulaR1C1 = "January"
    Range("C1").FormulaR1C1 = "February"
    Range("D1").FormulaR1C1 = "March"
End Sub
```

4. Insert a new blank worksheet, and test the macro.

The labels appear in the cells, and the original selection doesn't change.

Why should you get rid of *Select...Selection* pairs? One reason is that doing so does make the macro run faster. Another reason is that running a macro can seem less disruptive if it doesn't end with different cells selected than when it started. But the most important reason is unquestionably that *Select...Selection* pairs in a macro are a dead giveaway that you're a beginner who uses the macro recorder to create macros. It's OK to use the macro recorder; you just want to cover your tracks.

Simplify Select Groups

When you eliminate a Select...Selection pair, be sure that only a single statement uses the selection. If you have a single Select statement followed by two or more statements that use the selection, you can still avoid changing the selection, but you must do it in a different way.

1. In Excel, select a sheet with labels in the first row, and start recording a macro named **MakeBoldItalic**.
2. Click cell **B1**, click the **Bold** button, click the **Italic** button, and then click the **Stop Recording** button.



	A	B	C	D	E
1		January	February	March	
2					

3. Edit the macro to look like this:

```
Sub MakeBoldItalic()
    Range("B1").Select
    Selection.Font.Bold = True
    Selection.Font.Italic = True
End Sub
```

Obviously, if you delete the first Select...Selection pair, the macro won't control which cells will become italicized.

4. Edit the macro to assign the range to a variable named *myRange*. Then replace the **Selection** object with the **myRange** object.

The finished macro should look like this:

```
Sub MakeBoldItalic()
    Dim myRange As Range
    Set myRange = Range("B1")
    myRange.Font.Bold = True
    myRange.Font.Italic = True
End Sub
```

5. Change "B1" to "C1" in the macro, and then press F8 repeatedly to step through the macro. Watch how the format of the cell changes without changing which cell is originally selected.
6. Save the *Chapter04* workbook.

Eliminating the selection when there's a group might not seem like much of a simplification. And with only two statements, it probably isn't. But when you have several statements that use the same selection, storing the range in a variable can make the macro much easier to read.

Tip You could also replace the Select group with a With structure, like this:

```
With Range("B1")
    .Font.Bold = True
    .Font.Italic = True
End With
```

Secretly in the background, the With structure really just creates a hidden variable, takes the object from the With statement, and assigns that object to the hidden variable. It then puts the hidden variable in front of each “dangling” period. The End With statement discards the hidden variable. An advantage of using an explicit object variable is that you can declare the variable with a specific object type—for example, Dim myRange as Range—and then VBA checks to make sure any methods or properties you use are appropriate. With an explicitly declared variable, VBA also offers Auto Lists to help you modify a macro.

Entering Values and Formulas into a Range

You may have situations where you want to create a macro that dynamically enters formulas into cells. First you should understand how references work in formulas in Excel, and then you can see how to create formulas in a macro.

See Also This section refers to standard Excel formula references. For information about using structured formulas in a table, see the section titled “Record a Macro to Manipulate a Table” in Chapter 5, “Explore Data Objects.”

Relative References

Most formulas perform arithmetic operations on values retrieved from other cells. Excel formulas use cell references to retrieve values from cells. Imagine, for example, a list of Retail prices and Wholesale costs.

	A	B	C
1		Retail	Wholesale
2	High	5.50	2.75
3	Mid	4.50	2.25
4	Low	3.50	1.75
5			

Suppose you want to add a column to the list that calculates the *gross margin*—the difference between the Retail price and the Wholesale cost—for each item. You would put the label Margin in cell D1 and then enter the first formula into cell D2. The formula subtracts the first Wholesale cost (cell C2) from the first Retail price (cell B2). So you would enter =B2-C2 into cell D2.

	A	B	C	D	E
1		Retail	Wholesale	Margin	
2	High	5.50	2.75	2.75	
3	Mid	4.50	2.25		
4	Low	3.50	1.75		
5					

For each item in the High group, the gross margin is \$2.75. Now you need to copy the formula to the other rows. The formula you typed into cell D2 refers explicitly to cells C2 and B2. When you copy the formula to cell D3, you want the formula to automatically adjust to refer to C3 and B3. Fortunately, when you copy the formulas, Excel adjusts the references because, by default, references are relative to the cell that contains the formula. (The Prices worksheet in the *Chapter04* workbook contains these formulas.)

	A	B	C	D	E
1		Retail	Wholesale	Margin	
2	High	5.50	2.75	2.75	
3	Mid	4.50	2.25	2.25	
4	Low	3.50	1.75	1.75	
5					

If the reference =C2 is found in cell D2, it really means “one cell to my left.” When you copy the formula to cell D3, it still means “one cell to my left,” but now that meaning is represented by the reference =C3.

Absolute References

Sometimes you don’t want relative references. Imagine, for example, a worksheet that contains various quantities in column B and prices in row 3. (The Revenue worksheet in the *Chapter04* workbook contains the prices and quantities.)

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10						
5		20						
6		30						
7		40						
8		50						
9								

Suppose you want to add formulas to calculate the revenue for each combination. To calculate the first revenue value (cell C4), you need to multiply the first quantity (cell B4) by the first price (cell C3). When you type = B4*C3 into cell C4, you get the correct answer: \$50.

But if you copy that formula to cell C5, you get the ridiculous answer of \$1000. That's because the cell references are relative. In this version of the formula, you're not really referring to cells B4 and C3; you're referring to "one cell to my left" and "one cell above me." When you put the formula into cell C5, "one cell above me" now refers to cell C4, not cell C3.

		C5		fx =B5*C4	
	A	B	C	D	
1					
2			Price		
3		Quantity	\$5	\$10	
4		10	\$50		
5		20	\$1,000		
6		30			

In the Revenue table, you want the Quantity cell references to adjust from row to row, and you want the Price cell references to adjust from column to column, but you always want to reference the Quantity from column B and the Price from row 3. The solution in the user interface is to put a dollar sign (\$) in front of the B in the first Quantity reference (\$B4), and in front of the 3 in the first Price reference (C\$3). The formula that should go into cell C4 is =\$B4*C\$3. The dollar sign "anchors" that part of the formula, making it absolute. When you copy the formula to the rest of the range C4:E8, you get correct answers. (The RevenueFormulas worksheet in the *Chapter04* workbook contains the correct formulas.)

		E8		fx =\$B8*E\$3				
	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10	\$50	\$100	\$150			
5		20	\$100	\$200	\$300			
6		30	\$150	\$300	\$450			
7		40	\$200	\$400	\$600			
8		50	\$250	\$500	\$750			
9								

The relative portion of the formula changes with the row or column of the cell that contains the formula. The absolute portion remains fixed.

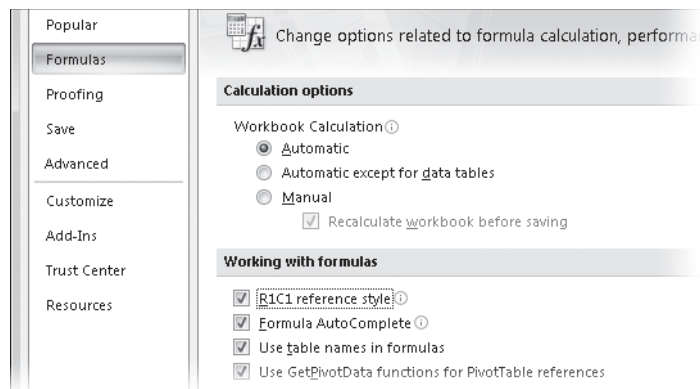
If you want to modify the formula so that it also takes into account the discount value from cell G3, you must make both the row and the column of the discount reference absolute. The correct formula would be =\$B4*C\$3*(1-\$G\$3). If you assign a name to a cell—for example, if you assign the name Discount to cell G3—then by default, using the name in the formula acts as a completely absolute reference. (The RevenueFormulas worksheet in the *Chapter04* workbook contains these formulas.) Later in this chapter, you will create a macro that will fill the grid with the correct formula, regardless of where it is on the worksheet and how many rows or columns it has.

R1C1 Reference Style

As a default, Excel displays letters for column headings and numbers for row headings. Consequently, the default name for the upper-left cell in the worksheet is cell A1. Referring to cells by letter and number is called *A1 reference style*. In A1 reference style, however, cell references do not really say what they mean. For example, the reference =C4 says “cell C4,” but if it’s in a formula in cell E4, it really means “two cells to my left,” and if it’s in a formula in cell C5, it really means “one cell above me.” You don’t know what the reference really means until you know which cell contains the reference.

Excel has an alternate reference style that uses numbers for both column and row headings. In this alternate reference style, to refer to a cell you use the letter *R* plus the row number and *C* plus the column number. Consequently, the upper-left cell in the worksheet is cell R1C1. Referring to cells by numbers in both rows and columns is called *R1C1 reference style*. In R1C1 reference style, cell references really do say what they mean. Consequently, in macros, when VBA has to understand and use the formulas, it is usually convenient to use R1C1 reference style. When a human has to understand the formula, it is usually easier to use A1 reference style, which is why A1 reference style is the default.

You can, however, change the user interface to use R1C1 reference style if you want to try it out. To turn on R1C1 reference style, click the Microsoft Office Button and then click Excel Options. On the Formulas page, select the R1C1 Reference Style check box, and click OK. (To turn off R1C1 reference style, clear the check box.) The setting in the Excel Options dialog box does not have any effect on macros: a macro can enter formulas using either reference style.



In R1C1 reference style, to specify a relative reference on the same row or column as the cell with the formula, you simply use an *R* or a *C*, without a number. For example, the reference =RC3 means “the cell in column 3 of the same row as me,” and the reference =R2C means “the cell in row 2 of the same column as me.”

To specify a relative reference in a different row or column, you indicate the amount of the difference, in square brackets, after the *R* or the *C*. For example, the reference =R5C[2] means “two columns to my right in row 5,” and the reference =R[-1]C means “one cell above me.”

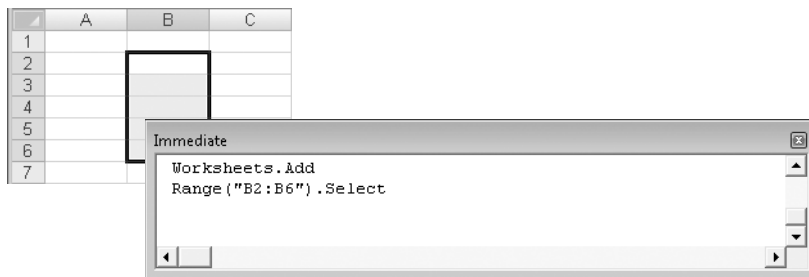
The correct formula for calculating the gross margin on the Prices worksheet was =B2-C2, but only if the formula was entered into cell D2. In R1C1 reference style, the equivalent formula is =RC[-2]-RC[-1], and it doesn't matter which row contains the formula. The formula to calculate the discounted price on the Revenue worksheet was =\$B4*C\$3*(1-\$G\$3), at least for cell C4. In R1C1 reference style, the same formula is =RC2*R3C*(1-R3C7), again, regardless of which cell contains the formula.

Important When you use A1 reference style, the formula changes depending on which range you copy the formula into. When you use R1C1 reference style, the formula is the same, regardless of which cell it goes into. The reference style only makes a difference when you put the same formula into multiple cells.

Put Values and Formulas into a Range

You can explore the properties for putting values and formulas into a range by creating a simple list of incrementing numbers.

1. In the Visual Basic editor, activate the **Immediate** window, type **Worksheets.Add**, and press Enter to create a new, blank worksheet in the active workbook.
2. Type **Range("B2:B6").Select** and press Enter to select a sample starting range of cells.



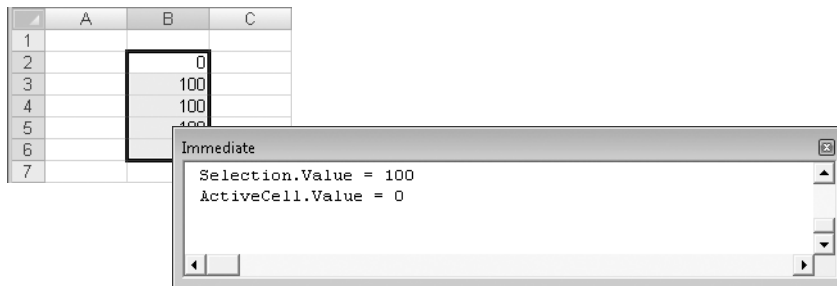
3. Type **Selection.Value = 100** and press Enter to fill all the cells of the selection with the number 100.

Value is a property of the range. When you set the Value property in conjunction with a multicell range, you change all the cells in the range.

Tip When assigning a constant value to a range, the Formula property is equivalent to the Value property, so `Selection.Formula = 100` is the same as `Selection.Value = 100`. The Formula property is equivalent to whatever you see in the formula bar when the cell is selected. The formula bar can contain constants as well as formulas, and so can the Formula property. When you assign a value to a cell, the Formula property and the Value property have the same effect.

4. Type `ActiveCell.Value = 0` and press Enter to change cell B2 to 0.

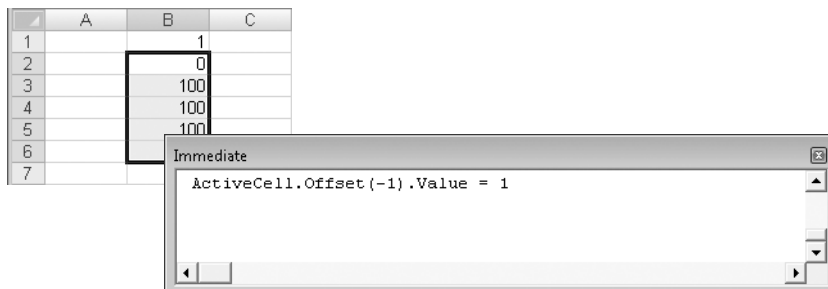
Only the active cell changes, not the selected cells. Entering a value in the active cell is equivalent to typing a value and pressing Enter. Entering a value in the selection is equivalent to typing a value and pressing Ctrl+Enter.



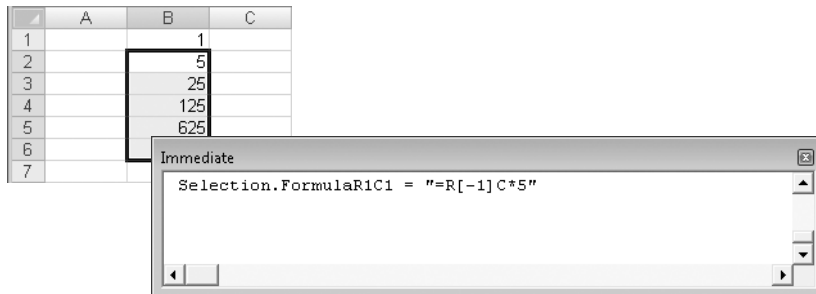
Suppose you want to enter a value in the cell above the active cell, whatever the active cell might be.

5. Type `ActiveCell.Offset(-1).Value = 1` and press Enter to change the value in cell B1 to 1.

This statement starts with the active cell, uses the Offset property to calculate a new cell one up from that starting cell, and then sets the Value property for the resulting cell.



6. Type `Selection.FormulaR1C1 = "=R[-1]C*5"` and press Enter.



Now each of the selected cells contains a formula, not a constant. The `FormulaR1C1` property expects a formula in R1C1 reference style. The reference `R[-1]C` always means “one cell above” regardless of which cell gets the formula.

7. Type `?ActiveCell.Value` and press Enter.

This statement displays the value 5 in the Immediate window. The `Value` property retrieves the result of any formula in a cell. When you retrieve the contents of the cell that contains a formula, the `Value` property gives you the result of the formula.

8. Type `?ActiveCell.Formula` and press Enter.

This statement displays the formula `=B1*5` in the Immediate window. When you retrieve the contents of a cell that contains a formula, the `Formula` property gives you the formula using A1 reference style. The setting in the Excel Options dialog box is ignored. If you want to retrieve the formula using R1C1 reference style, use the `FormulaR1C1` property.

All cells have `Formula`, `FormulaR1C1`, and `Value` properties. The `Value` property and the `Formula` property behave the same when you’re writing to the cell. When you read the value of a cell, the `Value` property gives you the value, and the `Formula` property gives you the formula using A1 reference style. The `FormulaR1C1` property is the same as the `Formula` property, except that it uses R1C1 reference style for all references, whether assigning a formula to the cell or reading the formula from a cell.

Tip The `Value` property always gives you the unformatted value of the number in a cell. A cell also has a `Text` property, which returns the formatted value of the cell. The `Text` property is read-only because it’s a combination of the `Value` property and the `NumberFormat` property. A range also has a `Value2` property. The difference between `Value` and `Value2` has to do with dealing with very large, very precise numbers—as in banking. The `Value` property uses a data type (double-precision floating point) that can handle either very large numbers or very precise numbers, but not at the same time. The `Value2` property uses a data type (currency) that can handle the large-scale precision needed in financial summaries.

Construct Formulas to Fill a Grid

Sometimes you need a macro to create formulas that contain references. For example, suppose you want to create a macro that will enter the appropriate formulas into the Revenue grid. You could just record a macro, but a recorded macro will use specific cell addresses. Suppose that the grid could be anywhere on the worksheet—not just starting in cell A1—and that it could be of any size. Your recorded macro can't handle that kind of variation.

If you can make a few simple assumptions, you can create a macro that will find a grid, select the current region to find the size of the grid, and then add the correct formula. You can even have the macro automatically find the location of the Discount cell and assign a name to it so that the formula can reference the cell by name. The assumptions you need to make are very useful for most simple macros:

- Always use the consistent words as labels so you can have the macro search for them.
- Always keep the same number of header columns and rows.
- Separate ranges are separated by at least one empty row or one empty column so that the CurrentRegion method can detect the rectangle.

On the Revenue worksheet, the searchable labels are Price and Discount, the grid has two header rows and one header column, and the ranges are separated by column F. With those simple assumptions, you can create a macro that will automatically create the right formula and put it into the correct range.

1. Make a copy of the Revenue worksheet in the *Chapter04* workbook, with cell A1 selected.

Copying the worksheet will give you a chance to test the macro, moving and resizing the revenue grid.

2. In the Visual Basic editor, enter the following macro shell, and press F8 twice to step to the End Sub statement.

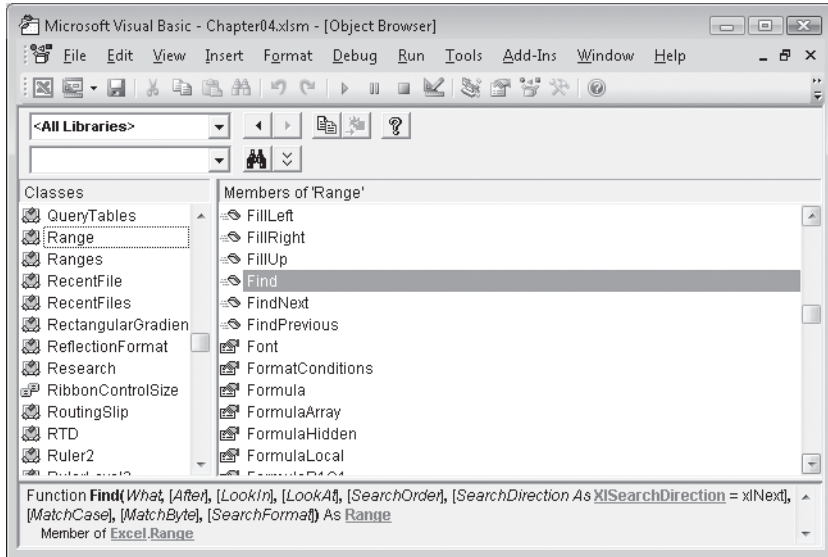
```
Sub FillFormulas()
    Dim myOuter as Range
    Dim myInner as Range
    Dim myFormula as String
```

```
End Sub
```

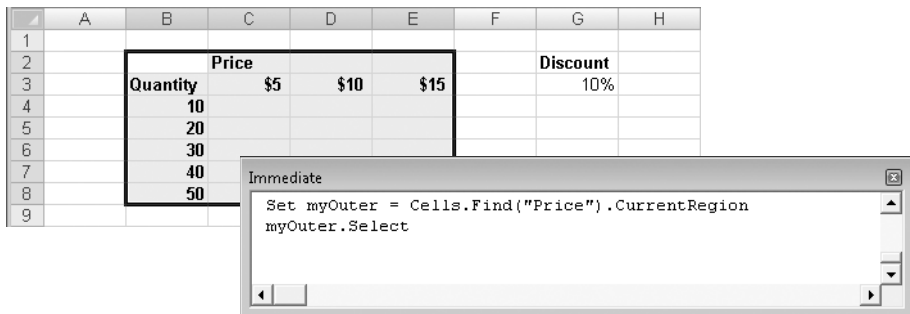
Declaring variables at the top will make it easier to work with different ranges. The myOuter range will refer to the entire current region of the Revenue grid, including the headings. The myInner range will refer to the empty cells in the middle that need formulas. The myFormula string will contain the formula so that you can construct the formula piece by piece in the macro.

3. In the **Object Browser**, select **Range** in the **Classes** list. Then, in the list of members, select the **Find** method.

The description indicates that this property has one required argument—the string you’re searching for—and that it returns a reference to a range.



4. In the **Immediate Window**, type `Set myOuter = Cells.Find("Price").CurrentRegion` and then press Enter. When you can confirm the correct range by entering `myOuter.Select`, copy the statement into the macro.



See Also The **Intersect** and **Offset** functions are described in the section titled “Refer to a Relative Range” earlier in this chapter.

For the `myInner` range, you need to remove two header rows at the top and the one header column at the left. You can do that by using the combination of **Intersect** and **Offset**.

5. In the **Immediate** window, type **Set myInner = Intersect(myOuter,myOuter.Offset(2,1))** and press Enter. When you can confirm the correct range by entering **myInner.Select**, copy the statement into the macro.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10						
5		20						
6		30						
7		40						
8		50						
9								

Immediate

```
Set myInner = Intersect(myOuter,myOuter.Offset(2,1))
myInner.Select
```

For the Discount range, you can use the label in the top cell to define the name for the lower cell. This creates a name in the worksheet, rather than a variable in Visual Basic. You assign the name by using the `CreateNames` method. The `CreateNames` method has four arguments, Top, Bottom, Left, and Right, respectively. These identify which side of the range contains the labels you should use as names. The Discount label is above the discount value, so Top is the only one you need to designate as True. Since Top is the first argument, you can simply omit the others.

6. In the **Immediate** window, type **Cells.Find("Discount").CurrentRegion.CreateNames True** and press Enter. When you can confirm the correct range by entering **Range("Discount").Select**, copy the statement into the macro.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10						
5		20						
6		30						
7		40						
8		50						
9								

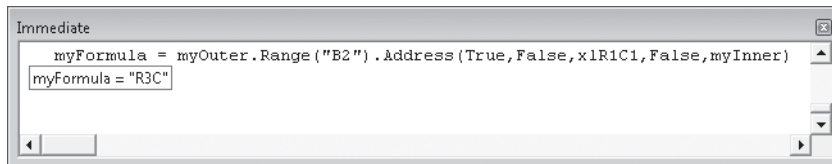
Immediate

```
Cells.Find("Discount").CurrentRegion.CreateNames True
Range("Discount").Select
```

For the first part of the formula, you need a reference to the first Price cell, which is currently cell C3. If you think of `myOuter` as if it were a worksheet, you want cell "B2" of that imaginary worksheet, and you want the address in R1C1 notation, with an

absolute row number and a relative column number, from the point of view of the first formula cell. The Address method gives you the address of a cell, with arguments to control what it looks like. Visual Basic prompts you for each of the arguments.

- In the **Immediate** window, type `myFormula = myOuter.Range("B2").Address(True,False,xlR1C1,False,myInner)` and press Enter. Move the mouse pointer over the word *myFormula* to confirm that the address is R3C, and then copy the statement into the macro.



Tip When you use the Range method, Visual Basic shows you tips for the methods and properties that follow. When you use the Cells method, Visual Basic does not show tips. Even though Range("B2") and Cells(2,2) are functionally equivalent, using the Range method makes the statement easier to type. If you find the Cells method easier to understand, you can make the change after you have successfully created the statement.

- In the **Immediate** window, type `myFormula = "=" & myFormula & "*"`. Move the mouse pointer over the word *myFormula* to confirm that the value is =R3C*, and then copy the statement into the macro.
- Enter the following three statements into the macro, optionally testing each one first in the **Immediate** window.

```
myFormula = myFormula & _
    myOuter.Range("A3").Address(False, True, xlR1C1, False, myInner)
myFormula = myFormula & " * ( 1 - Discount ) "
myInner.FormulaR1C1 = myFormula
```

There is nothing fundamentally new in these statements. The first one appends the quantity address, with relative column and absolute row. The second one adds the Discount portion of the formula. The Discount portion doesn't need to be converted to an address because it's already a name in the worksheet. The third statement assigns the finished formula to the inner range.

- Create a new copy of the Revenue worksheet, and test the macro. Then make another copy, change the size and location of the revenue grid, and test the macro again.

	A	B	C	D	E	F	G	H	I	J
1										
2		Discount								
3		10%								
4										
5					Price					
6				Quantity	\$5	\$10	\$15	\$20	\$25	
7				10	45	90	135	180	225	
8				20	90	180	270	360	450	
9				30	135	270	405	540	675	
10				40	180	360	540	720	900	
11				50	225	450	675	900	1125	
12				60	270	540	810	1080	1350	
13				70	315	630	945	1260	1575	
14				80	360	720	1080	1440	1800	
15				90	405	810	1215	1620	2025	
16				100	450	900	1350	1800	2250	
17										

Filling ranges of variable sizes with formulas is a powerful technique. You can use the methods and properties of the Range object to create the formula and to find the correct range to fill.

Formatting a Range

Formatting contributes much to the usability of a worksheet. Borders and background colors can emphasize parts of a report, and conditional formatting can highlight exceptions within a range. Cell formatting can be combined into cell styles to make the same formatting combinations easy to reuse.

Add Borders to a Range

Borders help to demarcate regions with a block of cells. Sometimes you want to put borders around every cell within a range. Sometimes you want to put a single border around an entire range of cells. Sometimes you want a different border along one side of a range. A Range object has methods and properties to allow you to completely control whatever type of border you need.

1. In Excel, make a copy of the **RevenueFormulas** worksheet. In the Visual Basic editor, copy the **TestRange** macro, give the new one the name **AddBorders**, and press F8 twice to initialize the myRange variable.

Troubleshooting If you don't have a TestRange macro, see the first two steps of the "Refer to a Relative Range" section earlier in this chapter.

2. In the **Immediate** window, type **Set myRange = Range("B2").CurrentRegion** and press Enter to assign the range containing the revenue calculations to the variable.

3. In the **Immediate** window, type `myRange.Borders.LineStyle =`.

As you type each period in the statement, an Auto List displays the available members. After you type the equal sign, no Auto List appears, but you can use the Object Browser to find the available options.

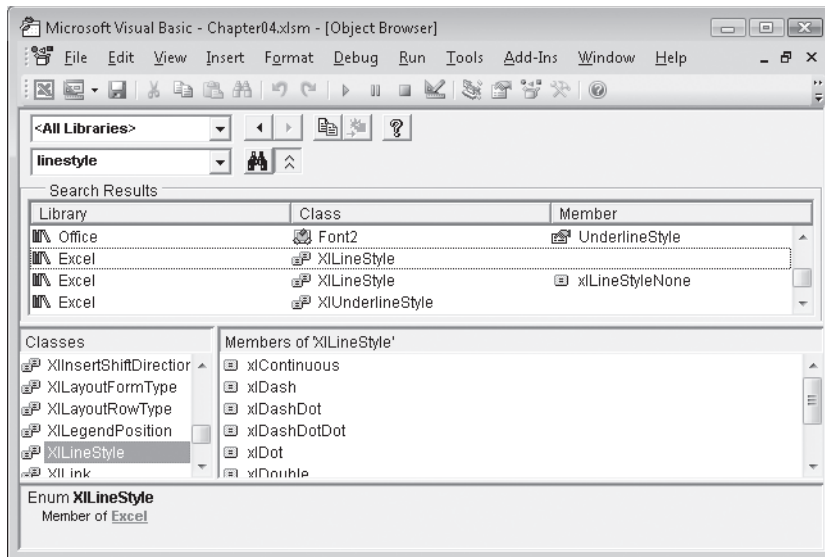


Search

4. In the **Object Browser**, in the **Search Text** box above the **Classes** list, type `LineStyle`, and click the **Search** button.

5. In the **Search Results** pane that appears, select `XILineStyle` in the **Class** list.

The **Member** list shows all the possible constants you can use for the `LineStyle` property.



`XILineStyle` is not really a class, even though it shows up in the list of classes in the Object Browser. There is no such thing as an `XILineStyle` object. It is, rather, an *enumerated list*. An enumerated list is used when a property or argument can accept only certain values. An enumerated list allows the object model designer to give each of those values a special name—for example, `xlContinuous`. Enumerated lists are included in the list of Classes, but with a special icon.

6. In the **Immediate** window, type `xlContinuous` to finish the statement, and then press Enter.

This adds a continuous border around each cell in the range. When you assign a value to the `LineStyle` property of the `Borders` object, the property changes for the border of each cell in the entire range.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10	45	90	135			
5		20	90	180	270			
6		30	135	270	405			
7		40	180	360	540			
8		50	225	450	675			
9								

Immediate

```
Set myRange = Range("E2").CurrentRegion
myRange.Borders.LineStyle = xlContinuous
```

7. In the **Immediate** window, type `myRange.Borders.LineStyle = xlNone` and press Enter to remove the borders.

The value `xlNone` does not appear in the enumeration list for `LineStyle` because it is a global constant that is used by many Excel objects. You can search for it in the Object Browser if you want to see the complete list of global constants.

The `Borders` object is actually a collection, and you can select specific borders within that collection. In principle, you could change cell borders one at a time, but because putting a border around an entire range is a common operation, there is a special method just for doing that.

8. In the **Immediate** window, type `myRange.BorderAround Weight:=xlThick` and press Enter.

This changes the edges of the range to a thick border. Because `Weight` is not the first argument, you have to type its name if you leave out `LineStyle`. Setting the border weight to `Thick` implies that the line will be continuous.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10	45	90	135			
5		20	90	180	270			
6		30	135	270	405			
7		40	180	360	540			
8		50						
9								

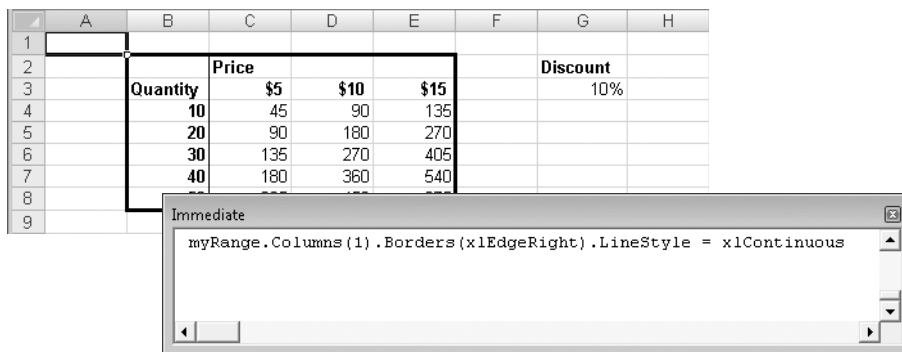
Immediate

```
myRange.Borders.LineStyle = xlNone
myRange.BorderAround Weight:=xlThick
```

Suppose that you want a border on the right side of the quantities. To specify a single border, you can use an enumerated name in conjunction with the `Borders` collection. Auto Lists can help you with the syntax, but you have to be a little tricky.

9. In the **Immediate** Window, type `myRange.Borders(xlEdgeRight).LineStyle = xlContinuous`, but do *not* press Enter. Immediately after `myRange`, type a period (.), type `Columns(1)`, and then press Enter.

Once you use the `Columns` property in a statement, you don't see any more Auto Lists, but if you temporarily leave out the `Columns` property, you get Auto Lists for everything else except the line style. Then, after you get the syntax correct for the statement, you can go back and add the `Columns` property.



10. In the **Immediate** window, type `myRange.Rows(2).Borders(xlEdgeBottom).LineStyle = xlContinuous` and press Enter. This adds a border under the row of prices.
11. Press F5 to end the macro. Copy the statements from the Immediate window into the `AddBorders` macro, and delete the two statements that fill and remove all the borders.

The finished macro should look like this:

```
Sub SetBorders()
    Dim myRange As Range
    Set myRange = Range("B2").CurrentRegion
    myRange.BorderAround Weight:=xlThick
    myRange.Columns(1).Borders(xlEdgeRight).LineStyle = xlContinuous
    myRange.Rows(2).Borders(xlEdgeBottom).LineStyle = xlContinuous
End Sub
```

12. Create a new copy of `RevenueFormulas` and test the finished macro.

`Borders` can emphasize parts of a report. The `Borders` collection allows you to change all the borders at one time or choose a particular type of border to modify. The `BorderAround` method is a convenient shortcut for assigning a border to all the edges of a multicell range.

Format the Interior of a Range

To enhance the readability of a worksheet, you might want to apply different background colors to various parts. For example, you might apply one format to all the cells that contain values that a user can input, and a different format to all cells that contain formulas.

1. In Excel, create another copy of the **RevenueFormulas** worksheet. In Visual Basic, copy the **TestRange** macro, name the new one **AddColors**, and press F8 twice to initialize the `myRange` variable.

Troubleshooting If you don't have a `TestRange` macro, see the first two steps of the "Refer to a Relative Range" section earlier in this chapter.

2. In the **Immediate** window, type `Set myRange = Range("B2").CurrentRegion` and press Enter to assign the range containing the revenue calculations to the variable.
3. In the **Immediate** window, type `myRange.Interior.Color =`.

As you type each period in the statement, an Auto List displays the available members. After you type the equal sign, however, no Auto List appears. For the `Color` property, there is no enumerated list. You can enter any number between 0 (which equals black) and 16777215 (which equals white), so there are literally more than 16 million possible values. This is a major change from previous versions of Excel, where colors in a worksheet were limited to a palette of only 56 colors.

Colors on a computer correspond to the red, green, and blue guns of a cathode ray tube. (Liquid crystal displays use a different technology, but the same component colors.) Visual Basic has an RGB function you can use to specify precise red, green, and blue components, but Excel provides an easier way to specify the color you want: it includes an enumerated list that gives meaningful names to about 140 of the most common colors.

See Also [Excel 2007 also uses theme colors to help you use predefined sets of compatible colors. Theme colors are described in more detail in the section titled "Add a Gradient Fill to a Cell" in Chapter 6, "Explore Graphical Objects."](#)

4. In the **Immediate** window, type `rgbMediumVioletRed` to complete the statement, and press Enter. (Once you get past `rgbM`, press Ctrl+Space to get to the middle of the `rgb` color values.)

The background color of the entire range changes to a medium violet red.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10	45	90	135			
5		20	90	180	270			
6		30	135	270	405			
7		40	180	360	540			
8		50						
9								

Immediate

```
Set myRange = Range("B2").CurrentRegion
myRange.Interior.Color = rgbMediumVioletRed
```

Now that Excel can handle millions of colors, it has a new capability to change how light (the tint) or dark (the shade) a color is without changing the actual color (the hue).

- In the **Immediate** window, type `myRange.Interior.TintAndShade = -0.2` and press Enter. The color changes to a slightly darker shade of violet red.

A range object has a special method called *SpecialCells* that isolates cells within the range based on various attributes. For example, you can reference all the formula cells within the range.

- In the **Immediate** window, type `myRange.SpecialCells(xlCellTypeFormulas).Interior.TintAndShade = 0.3` and press Enter.

The block of formulas changes to a lighter tint of violet red. In this range, the formulas form a contiguous block, but *SpecialCells* can return a range of discontinuous cells as well.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10	45	90	135			
5		20	90	180	270			
6		30	135	270	405			
7		40	180	360	540			
8								
9								

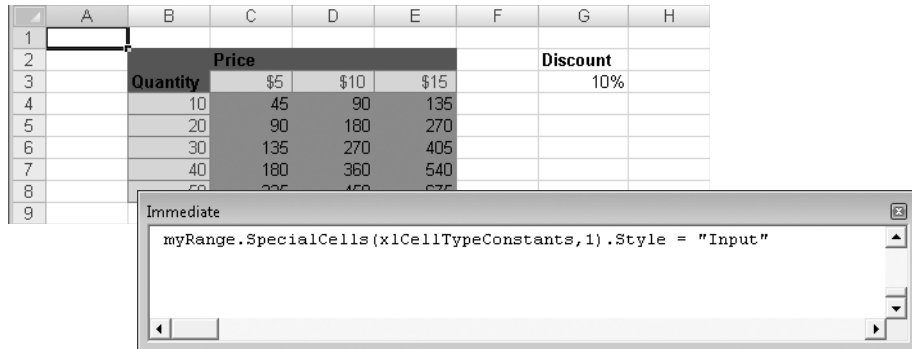
Immediate

```
myRange.Interior.TintAndShade = -0.2
myRange.SpecialCells(xlCellTypeFormulas).Interior.TintAndShade = 0.3
```

In Excel, you can give a name to a set of formatting characteristics. This is called a *cell style*. There are several built-in styles in a workbook. One of them is named *Input*, so that you can use it to format cells that can accept user input—typically cells that contain constants that are numbers.

7. In the **Immediate** window, type `myRange.SpecialCells(xlCellTypeConstants, xlNumbers).Style = "Input"` and press Enter.

The cells with prices and quantities change to a light tan, with borders around each cell. The constant `xlNumbers` doesn't appear in an Auto List, but you can find the list in the Object Browser by searching for `SpecialCells`.



You can modify the style format in the same way that you can modify a range format directly.

8. Enter the following two statements in the **Immediate** window:

```
ActiveWorkbook.Styles("Input").Interior.Color = rgbMediumVioletRed
ActiveWorkbook.Styles("Input").Interior.TintAndShade = 0.5
```

This changes the Input style so that it has a lighter version of the same violet red shade as the rest of the cells. When applying a style to cells that can take input values, you may want to search the entire worksheet for the numeric constants. To do that, you just start with the global `Cells` property.

9. In the **Immediate** window, type `Cells.SpecialCells(xlCellTypeConstants, xlNumbers).Style = "Input"` and press Enter.

This adds the Input style to the Discount cell value. If you had hundreds of input cells scattered all over the worksheet, this statement would still find them all. The text labels in the Revenue range are hard to read, with the black text on a dark background. You can use `SpecialCells` to isolate all the cells that contain text constants.

10. In the **Immediate** window, type `myRange.SpecialCells(xlCellTypeConstants, xlTextValues).Font.Color = rgbWhite` and press Enter.

This changes the font color for the labels to white, but they would look better bold as well. In fact, all the constants within the formula range would look better if they were bold.

11. In the **Immediate** window, type `myRange.SpecialCells(xlCellTypeConstants).Font.Bold = True` and press Enter.

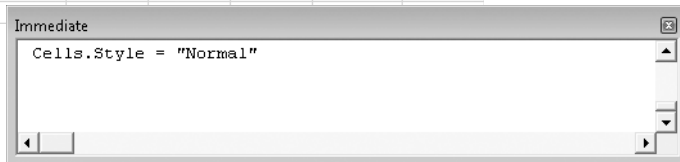
This changes all the constants within the range store in `myRange` to bold. By leaving out the second argument to `SpecialCells`, you get everything that matches the general type. You can also use a special style to clear all the formatting.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	\$5	\$10	\$15		10%	
4		10	45	90	135			
5		20	90	180	270			
6		30	135	270	405			
7		40	180	360	540			
8		50	225	450	675			
9								

12. In the **Immediate** window, type `Cells.Style = "Normal"` and press Enter.

This clears all the formatting, including the number and formats. When you clear formats from a worksheet, what it really does is apply the Normal style to all the cells. By changing the Normal style, you change the default appearance of cells in the workbook.

	A	B	C	D	E	F	G	H
1								
2			Price				Discount	
3		Quantity	5	10	15		0.1	
4		10	45	90	135			
5		20	90	180	270			
6		30	135	270	405			
7		40	180	360	540			
8		50	225	450	675			
9								



13. Press F5 to end the macro. Copy the statements from the **Immediate** window into the **AddColors** macro, and delete the statement that clears all the formatting.

The finished macro, ignoring optional line breaks, should look like this:

```
Sub SetColors()
    Dim myRange As Range
    Set myRange = Range("B2").CurrentRegion

    myRange.Interior.Color = rgbMediumVioletRed
    myRange.Interior.TintAndShade = -0.2
    myRange.SpecialCells(xlCellTypeFormulas). _
        Interior.TintAndShade = 0.3
```

```

myRange.SpecialCells(xlCellTypeConstants, xlNumbers). _
    Style = "Input"
ActiveWorkbook.Styles("Input").Interior _
    .Color = rgbMediumVioletRed
ActiveWorkbook.Styles("Input").Interior _
    .TintAndShade = 0.5
Cells.SpecialCells(xlCellTypeConstants, xlNumbers) _
    .Style = "Input"

myRange.SpecialCells(xlCellTypeConstants, xlTextValues) _
    .Font.Color = rgbWhite
myRange.SpecialCells(xlCellTypeConstants).Font.Bold = True
End Sub

```

14. Create a new copy of **RevenueFormulas** and test the finished macro.

Ranges are powerful objects. They are the essence of Excel. With ranges you can organize information, create formulas, and apply formatting. And you can do all of that with under the control of VBA macros.



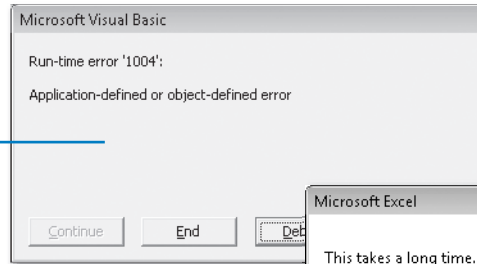
CLOSE the *Chapter04.xlsm* workbook.

Key Points

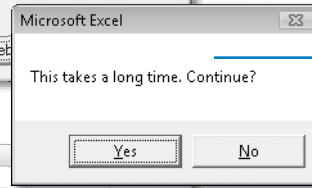
- Use the Object Browser to find out what members—methods and properties—are available for an object, and what each method or property returns.
- Avoid changing the selection during your macros. A macro runs faster and appears more professional if it doesn't have to repaint the screen.
- While debugging, use the Immediate window to test the current reference of a range object.
- Many range-related functions start with one range and return another range. These functions are invaluable for navigating from one range to another. The most important one is `CurrentRegion`.
- Always use `R1C1` references when constructing formulas from macros, and take advantage of the many options—relative, absolute, internal, external—that the `Address` property gives you.
- Use the `Borders` collection to simultaneously control the borders of each cell within a range. Use the `BorderAround` method to treat the range as a single unit.
- Use either the `RGB` function or the enumerated list of RGB constants to select a color. To create smooth gradations of the shades and tints of a color, take advantage of the `TintAndShade` property.

Chapter at a Glance

Prevent errors by using an If statement, **page 210**



Ask a question by using a message box, **page 217**



	A	B	C	D	E	F
1						
2	Old				New	
3	143	116	110		121	121 142
4	133	136	114		132	146 106
5	123	113	120		149	145 128
6	103	148	129		110	119 113
7						

Loop over parallel ranges by using a For loop, **page 222**

	A	B
1	EIS.xlsx	
2	Flow.xlsx	
3	Graphics.xlsx	
4	Loan.xlsx	
5	Orders.xlsx	
6	Ranges.xlsx	
7	Budget.xlsx	
8		

Loop indefinitely by using a Do loop, **page 225**

```
myStop = Range("A1").CurrentRegion.Rows.Count
For myRow = 3 To myStop
    Application.StatusBar = "Processing row " & myRow & " of " & myStop
    If Cells(myRow, 1) <> Cells(myRow + 1, 1) Then
        Cells(myRow, 1).Select
        ActiveCell.PageBreak = xlPageBreakManual
    End If
    Next myRow
Cells(myRow,
```

Debug large loops by using a break-point, **page 229**

527	Dinosaurs	January-05	Washington	Retail
528	Dinosaurs	February-05	Washington	Retail
529	Dinosaurs	February-05	Washington	Wholesale

Processing row 1046 of 3266

Show progress by using the status bar, **page 233**

7 Control Visual Basic

In this chapter, you will learn to:

- ✓ Use conditional statements.
 - ✓ Create loops using three different blocks.
 - ✓ Retrieve the names of files in a folder.
 - ✓ Create breakpoints to debug long loops.
 - ✓ Show progress while a macro executes a loop.
-

The first successful underwater tunnel ever built was begun in 1825. It is the Thames Tunnel. It was a financial disaster at the time, but amazingly it is still in use as part of the London Underground system. The genius behind the the tunnel's engineering was a man named Marc Brunel. Twenty years before launching the Thames Tunnel, Brunel made a name for himself by devising a way of inexpensively producing the pulley blocks needed to build ships for the British shipping industry. Brunel's technique later came to be known as an "assembly line," and Henry Ford turned the invention into an industry, supplying America with Model T cars that cost only \$3,500 in today's dollars.

Repetition can have a dramatic effect on efficiency. Computer programs—including macros that you write—become more powerful when you add a multiplier effect. In this chapter, you'll learn how to add loops to your macros. And to make those loops more effective, you'll learn how to create conditional expressions that let the macro make decisions.



Important Before you complete this chapter, you need to install the practice files from the book's companion CD to their default locations. See "Using the Book's CD" on page xv for more information.



USE the *Flow.xlsx* workbook, the *Flow.txt* text file, and the *Orders.xlsx* workbook. These practice files are located in the *Documents\MSP\Excel\VBA07SBS* folder. The *Flow* text file contains some initial macros that you will copy into your workbook and modify during this chapter. The initial macros are stored in a simple text file so that you can be certain there is no malicious code before you put the code into a trusted location.

BE SURE TO save the *Flow.xlsx* workbook as a macro-enabled workbook named *Chapter07.xlsm* in the trusted location you created in Chapter 1.

OPEN the *Flow* text file. Then open the *Chapter07* workbook, right-click any sheet tab, and click View Code to open the Microsoft Visual Basic editor. In the Visual Basic editor, from the Insert menu, click Module to create a new module for your macros, and then save the file. Arrange the Microsoft Office Excel 2007 and Visual Basic editor windows so that you can see both of them side by side.

Using Conditionals

Recorded macros are not very smart. They can repeat what you did when you recorded the macro, but they can't behave differently in different circumstances. They can't make decisions. The only way that you can make your macros "smart" is to add the decision-making ability yourself.

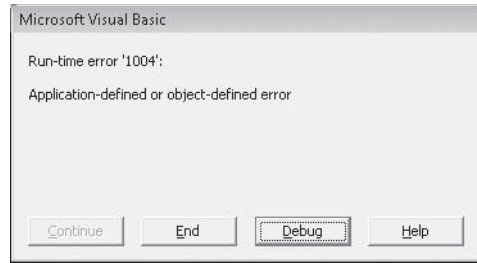
Make a Decision

The *Flow* text file contains a macro named *MoveRight*, which looks like this:

```
Sub MoveRight()  
    ActiveCell.Offset(0, 1).Select  
End Sub
```

This macro selects the cell to the right of the active cell and works fine—most of the time.

1. Copy the **MoveRight** macro from the text file, and paste it into a VBA module in the *Chapter07* workbook.
2. With cell **A1** selected in the workbook, activate the Visual Basic editor, click in the **MoveRight** macro, and press F5.
The macro selects cell B1 in the workbook.
3. In Excel, press Ctrl+Right Arrow to select cell **XFD1**, the rightmost cell on the first row.
4. In the Visual Basic editor, press F5.
Visual Basic displays an error.



You can't select the cell to the right of the rightmost cell. If your macro can't move to the right, you'd rather have it do nothing than display an error message.

5. In the error message box, click the **Debug** button to jump to the macro, and then click the **Reset** button to stop the macro.
6. Insert the statement **If ActiveCell.Column < Columns.Count Then** after the **Sub** statement. Indent the statement that changes the selection, and then insert the statement **End If** before the end of the macro.

Be sure to indent each statement in such a way as to make it clear which statement is governed by the If statement. Visual Basic doesn't require proper indentation, but indentation is critical to help you (or someone following after you) interpret the macro the same way that Visual Basic does.

The revised macro should look like this:

```
Sub MoveRight()
    If ActiveCell.Column < Columns.Count Then
        ActiveCell.Offset(0, 1).Select
    End If
End Sub
```

An If statement (a statement that begins with the word *If*) pairs with an End If statement. The group of statements from the If to the End If is called an If block.

Visual Basic looks at the expression immediately after the word *If* and determines whether it evaluates to True or False. This true-or-false expression is called a conditional expression. In a simple If block such as this example, if the value of the expression is True then Visual Basic executes all the statements between the If statement and the End If statement. If the expression is False, Visual Basic jumps directly to the End If statement. You must always put the word *Then* at the end of the If statement. In this case, the conditional expression tests for whether the current column is less than the total number of columns in the worksheet. You could also compare to a constant—such as 16384 or 2^{14} —but using object properties allows the macro to work with older versions of Excel (with 256 columns) and also with Excel 2007 (with 16384 columns).

7. Switch back to Excel, select cell **XFA1**, activate Visual Basic, and then press F5 four or five times.

The macro moves the active cell to the right until it gets to the last cell. After that it does nothing, precisely according to your instructions.

XEZ	XFA	XFB	XFC	XFD

The macro recorder will never create an *If* block. This kind of decision is pure Visual Basic, and you must add it yourself. Fortunately, adding an *If* block is easy.

1. Figure out a question that has a “yes” or “no” answer. In this example, the question is, “Is the column number of the active cell less than 256?” You can turn this question into the true-or-false conditional expression in an *If* statement.
2. Put the word *If* in front of the conditional expression, and put the word *Then* after it.
3. Figure out how many statements you want to execute if the conditional expression returns a True value.
4. Put an *End If* statement after the last statement that you want controlled by the *If* block.

By using *If* blocks, you can add intelligence to your macros.

Make a Double Decision

Sometimes—such as when you’re preventing an error—you want your macro to execute only if the conditional expression is True. Other times, you want the macro to behave one way if the expression is True and a different way if the condition is False.

For example, suppose that you want a macro that moves the active cell to the right, but only within the first five columns of the worksheet. When the active cell gets to the fifth column, you want it to move back to the first cell of the next row. In this case, you want the macro to carry out one action if the cell column is less than five (move to the right) and a different action if it isn’t (move down and back). You can make the macro choose between two options by adding a second part to the *If* block.

1. Switch to the Visual Basic editor, and copy the **MoveRight** macro. Change the name of the new copy to **FiveColumnWrap**.
2. In the **FiveColumnWrap** macro, change the expression **Columns.Count** to **5** in the *If* statement.

3. Add the statement **Else** before the End If statement, and press Enter.
4. Press Tab, and add the statement **Cells(ActiveCell.Row + 1, 1).Select** after the Else statement.

The revised macro should look like this:

```
Sub MoveRight()
    If ActiveCell.Column < 5 Then
        ActiveCell.Offset(0, 1).Select
    Else
        Cells(ActiveCell.Row + 1, 1).Select
    End If
End Sub
```

The Else statement simply tells Visual Basic which statement or statements to execute if the conditional expression is False.

Tip Several different statements would select the first cell of the next row. For example, here are a few alternatives:

```
Rows(ActiveCell.Row + 1).Cells(1).Select
```

```
ActiveCell.EntireRow.Cells(2, 1).Select
```

```
ActiveCell.Offset(1, 0).EntireRow.Cells(1).Select.
```

They all get from the same starting point (the ActiveCell) to the same destination. When you write macros, you often have multiple alternatives. You simply choose the one that is easiest to understand.

5. Press F5 repeatedly to execute the macro.

You see the selection move to the right and then scroll back to column A, much as a word processor wraps to the next line.

	A	B	C	D	E	F
1						
2						
3						
4						

An If block can contain a single part, executing statements only when the conditional expression is True, or it can have two or more parts, executing one set of statements when the conditional expression is True and a different set when it's False.

Tip In most cases, If and Else are sufficient. There is also a way to use an If block to create multiple conditions by adding an Elself statement. To find out more about If blocks, highlight the word If in the macro and then press F1.

Ask Yourself a Question

In Chapter 2, “Make a Macro Do Complex Tasks,” you created a macro that asked you to enter a date. You used the Visual Basic `InputBox` function to do that. The `InputBox` function is excellent for asking a question, but you must be careful about what happens when you click the `Cancel` button.

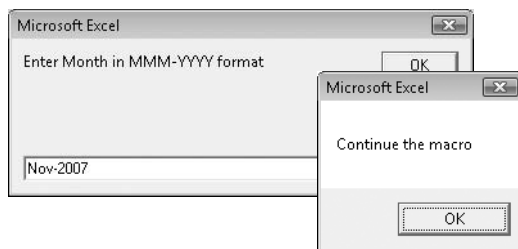
The *Flow* text file contains a macro named *TestInput* that prompts for the date. The code in this macro should look familiar.

```
Sub TestInput()  
    Dim myDate As String  
    myDate = InputBox("Enter Month in MMM-YYYY format")  
    MsgBox "Continue the macro"  
End Sub
```

The macro prompts for a date. It then displays a simple message box indicating that it's running the rest of the macro.

1. Copy the **TestInput** macro from the text file, and paste it into a module in the *Chapter07* workbook in the Visual Basic editor.
2. Click in the **TestInput** macro. Press F5 to run the macro, type **Nov-2007** for the date, and then click **OK**.

The message box appears, simulating the rest of the macro.



3. Click **OK** to close the message box.
4. Press F5 to run the macro again, but this time click **Cancel** when prompted to enter the date.

The message box still appears, even though your normal expectation when you click `Cancel` is that you'll actually cancel what you started.

- Click **OK** to close the message box.

You need a conditional expression where a True result means that you want the macro to continue. An appropriate question is, "Did the user enter anything in the box?" since clicking Cancel is the same as leaving the box empty: Whether you click Cancel or leave the box empty, the `InputBox` function returns an empty string (equivalent to two quotation marks with nothing between them). The operator `<>` (a less-than sign followed by a greater-than sign) means "not equal;" it's the opposite of an equal sign.

- Before the **MsgBox** statement, enter the statement `If myDate <> "" Then`. Before the **End Sub** statement, enter `End If`. Indent the statement inside the **If** block.

The revised macro should look like this:

```
Sub TestInput()
    Dim myDate As String
    myDate = InputBox("Enter Month in MMM-YYYY format")
    If myDate <> "" Then
        MsgBox "Continue the macro"
    End If
End Sub
```

- Press F5 and test to make sure the macro properly handles an input value. Type a date, and click **OK**.

The macro "continues."

- Click **OK** to close the message box.
- Now run the macro again, but this time click **Cancel** when prompted for a date.

The macro stops quietly.

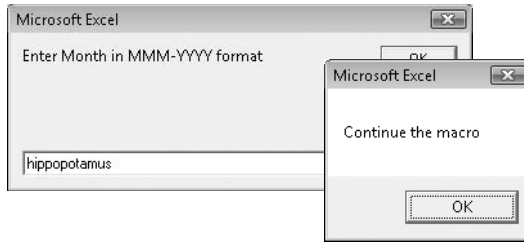
Whenever you allow user input in a macro, you must be sure to check whether the user took the opportunity to cancel the macro entirely.

Test for a Valid Entry

Testing for an empty string checks to see whether the user clicked the Cancel button, but it does not help you determine whether the value entered into the box is valid. You can add a second test to check the input value.

- Run the **TestInput** macro again, but this time type **hippopotamus** in the input box, and click **OK**.

The macro continues—the same as it would have if you had entered a date.



2. Click **OK** to close the message box.

This behavior could be a problem. You need to check whether the box is empty, but you also need to check for a valid date. Visual Basic has an `IsDate` function that will tell you whether Visual Basic can interpret a value as a date. However, you want to check for a date only if the user didn't click **Cancel**. This calls for nested `If` blocks.

3. Change the macro to look like this:

```
Sub TestInput()
    Dim myDate As String
    myDate = InputBox("Enter Month in MMM-YYYY format")
    If myDate <> "" Then
        If IsDate(myDate) Then
            MsgBox "Continue the macro"
        Else
            MsgBox "You didn't enter a date"
        End If
    End If
End Sub
```

Be sure to indent each statement in such a way as to make it clear which statement is governed by which `If` or `Else` statement.

4. Run the macro at least three times. Test it with a valid date, with an invalid entry, and by clicking **Cancel**.

The valid and invalid entries should display the appropriate messages. Clicking **Cancel** or leaving the box empty should display no message.

Tip Visual Basic can interpret several different formats as dates. Try different date formats, such as 11/07, to see which ones Visual Basic interprets as dates.

Using the `InputBox` function can be a valuable way of making a macro useful across a wide range of circumstances. You must be careful, however, to check the result of the `InputBox` before you continue the macro. Typically, you need to check for three possibilities: valid input, invalid input, and **Cancel**. An `If` block—and sometimes a nested `If` block—can make your macro smart enough to respond to all the possible options.

Ask with a Message

The Visual Basic MsgBox function is handy for displaying simple messages. As its name implies, this function displays a message box. The MsgBox function can do much more than that, however. It can ask questions, too. Many times, when a macro asks a question, all it needs is a simple “yes” or “no” answer. The MsgBox function is perfect for yes-or-no questions.

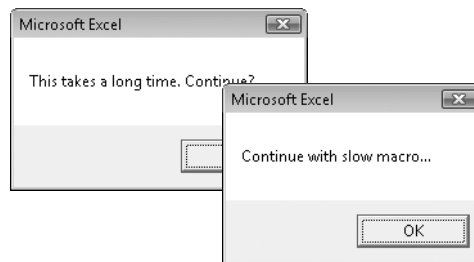
Suppose that you have two macros. One is a long, slow macro named *PrintMonth*, and the other is a short, quick macro named *ProcessMonth*. You find that you often accidentally run the slow one when you intend to run the quick one. One solution might be to add a message box to the beginning of the slow macro that asks you to confirm that you intended to run the slow one.

The *Flow* text file includes a macro named *CheckRun*. You’ll enhance this macro to see how to use a MsgBox function to ask a question. The macro looks like this before you start:

```
Sub CheckRun()  
    MsgBox "This takes a long time. Continue?"  
    MsgBox "Continue with slow macro..."  
End Sub
```

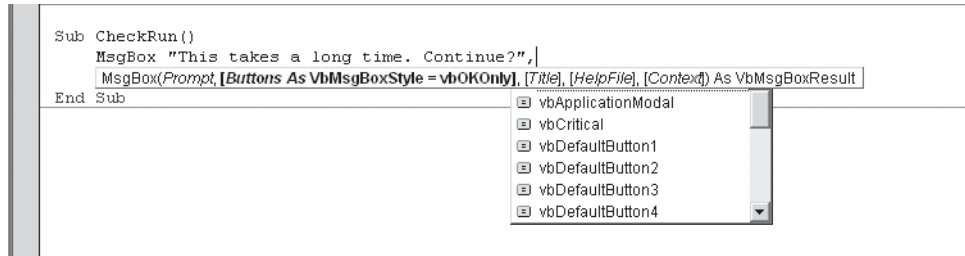
1. Copy the **CheckRun** macro from the text file into a module in the *Chapter07* workbook.
2. Click in the **CheckRun** macro, and press F5 to run it. Click **OK** twice to close each message box.

The first message box appears to ask a question, but it has only a single button. To ask a question, you must add more buttons.



3. Move the cursor to the end of the first **MsgBox** statement. Immediately after the closing quotation mark, type a comma.

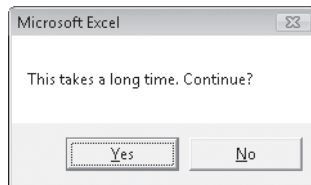
As soon as you type the comma, Visual Basic displays the Quick Info for the `MsgBox` function. The first argument is named *Prompt*. That's the one in which you enter the message you want to display. The second argument is named *Buttons*. This is an enumerated list of values. The default value for *Buttons* is `vbOKOnly`, which is why you saw only a single OK button when you ran the macro before.



Along with the Quick Info box, Visual Basic also displays the Auto List of possible values for the *Buttons* argument. You want the buttons to ask the question in terms of yes or no.

4. Scroll nearly to the bottom of the list, select `vbYesNo`, press Tab, and then press F5 to run the macro.

The first message box now has two buttons.



5. Click **Yes** to close the first message box, and then click **OK** to close the second one.

The message box asks a question, but it totally ignores your answer. You need to get the answer from the `MsgBox` function and use that answer to control the way the macro runs.

6. Type the statement `Dim myCheck As VbMsgBoxResult` at the beginning of the macro.

When you know a variable will contain only the value from an enumerated list, you can use the name of the list when you declare the variable. When you later write a statement to test the value of the variable, Visual Basic will display the list of possible values for you.

7. At the beginning of the first `MsgBox` statement, type `myCheck =` and then put parentheses around the argument list of the `MsgBox` function.

The revised statement should look like this:

```
myCheck = MsgBox("This takes a long time. Continue?", vbYesNo)
```

Important When you use the return value of a function such as `MsgBox`, you must put parentheses around the argument list. When you don't use the return value, you must not use parentheses.

8. Insert these three statements before the second `MsgBox` statement:

```
If myCheck = vbNo Then
    Exit Sub
End If
```

Important When you create a conditional expression using the result of the `MsgBox` function, you must not check for `True` or `False`. `MsgBox` has many types of buttons it can display, so it has many types of answers. If you use `vbYesNo` for the `Buttons` argument, `MsgBox` will always return either `vbYes` or `vbNo`. Neither of these enumerated values equals `False`, so comparing the result to `False` would be the same as always clicking `Yes`. When you test for a value that comes from an enumerated list, always be sure to use the appropriate enumeration constant.

The `Exit Sub` statement causes Visual Basic to stop the current macro immediately. To avoid making your macros hard to understand, you should use `Exit Sub` sparingly. One good use for `Exit Sub` is when you cancel the macro at the beginning, as in this case. The finished macro should look like this:

```
Sub CheckRun()
    Dim myCheck As VbMsgBoxResult

    myCheck = MsgBox("This takes a long time. Continue?", vbYesNo)
    If myCheck = vbNo Then
        Exit Sub
    End If

    MsgBox "Continue with slow macro..."
End Sub
```

9. Test the macro. Run it and click **Yes**, and then run it and click **No**. Make sure the rest of the macro runs only when you click **Yes**.

A message box is a powerful tool for asking simple questions. The `MsgBox` function is also a good example of how to use parentheses around argument lists: use parentheses if you use the return value of the function; otherwise, don't use them.

Creating Loops

Long before Henry Ford, and even before Marc Brunel, the economist Adam Smith reasoned that in a single day, a single worker could make only one straight pin, but ten people could subdivide the work and create 48,000 pins in the same day—an almost 5,000-fold increase in productivity. Similarly, you can get amazing increases in productivity by converting a macro that runs once into one that runs thousands of times in a loop.

Loop Through a Collection by Using a For Each Loop

Excel allows you to protect a worksheet so that users can change only cells that are explicitly unlocked. You must, however, protect each sheet individually. Suppose that you have a workbook containing budgets for ten different departments and that you want to protect all the worksheets.

The *Flow* text file includes a macro named *ProtectSheets*. Here's what it looks like:

```
Sub ProtectSheets()
    Dim mySheet As Worksheet
    Set mySheet = Worksheets(1)
    mySheet.Select
    mySheet.Protect "Password", True, True, True
End Sub
```

This macro assigns a reference to the first worksheet to the `mySheet` variable, selects that sheet, and then protects it. (Selecting the sheet really isn't necessary, but it makes it easier to see what the macro is doing.) Now see how you can convert this macro to protect all the worksheets in the workbook.

1. Copy the **ProtectSheets** macro from the text file, and paste it into a VBA module in the *Chapter07* workbook.
2. Click in the **ProtectSheets** macro, and press F8 repeatedly to step through the macro. Make sure you understand everything that the original macro does.
3. In the third line, replace **Set** with **For Each**, replace the equal sign with **In**, and remove the parentheses and the number between them.
4. Indent the two statements that begin with **mySheet**, add a new line, and then type the statement **Next mySheet**.

The finished macro should look like this:

```
Sub ProtectSheets()
    Dim mySheet As Worksheet
    For Each mySheet In Worksheets
        mySheet.Select
    Next mySheet
End Sub
```



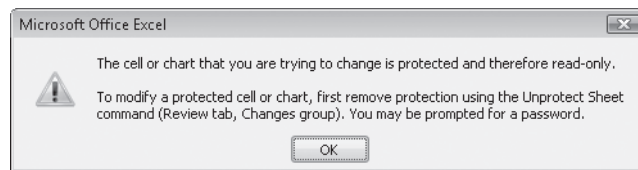
```

        mySheet.Protect "Password", True, True, True
    Next mySheet
End Sub

```

The For Each statement acts just like Set: It assigns an object reference to a variable. But instead of assigning a single object to the variable, it assigns each item from a collection to the variable. Then, for each (get it?) object in the collection, Visual Basic executes all the statements down to the Next statement. (Technically, you don't need to put the variable name after Next. If you do use it, Visual Basic requires that it match the variable name after For Each. Always use the loop variable after Next so that Visual Basic can help you avoid creating bugs in your macros.) Statements beginning with For Each and ending with Next are called For Each blocks or For Each loops.

5. Press F8 repeatedly to step through the macro, watching as it works on each worksheet in turn.
6. Switch to Excel, and try typing a value into a cell on any worksheet. Afterwards, close the error message box that opens.



7. Create a new macro named **UnprotectSheets** that unprotects all the worksheets. Try to write the macro without looking at the finished code that follows. Hint: You'll need to use the Unprotect method of the worksheet object, with a single argument that gives the password.

Tip A For Each loop is a handy way of browsing collections in the Immediate window. However, in the Immediate window, everything you type must be on a single line. You can put multiple statements on a single line by separating the statements with colons. For example, here's what you'd type in the Immediate window to see the names of all the worksheets in the active workbook: *For Each x In Worksheets: ?x.Name: Next x*. (In the Immediate window, it's all right to use short, meaningless names for variables.)

Here's what the UnprotectSheets macro should look like:

```

Sub UnprotectSheets()
    Dim mySheet As Worksheet
    For Each mySheet In Worksheets
        mySheet.Select ' This statement is optional.
        mySheet.Unprotect "Password"
    Next mySheet
End Sub

```

8. Save the workbook, press F5 to run the **UnprotectSheets** macro, and then test it by changing a value on a worksheet.

Looping through a collection is almost as easy as assigning a single object to a variable. The only differences are that you use For Each instead of Set, you specify a collection to loop through, and you add a Next statement to end the loop.

Loop with a Counter by Using a For Loop

Sometimes you want to perform actions repeatedly but can't use a For Each loop. For example, a For Each loop can work through only a single collection. If you want to compare two parallel collections—such as two ranges—you can't use a For Each loop. In that situation, Visual Basic has another, more generalized way to loop: a For loop.

The Compare worksheet in the *Chapter07* workbook contains two named ranges. The one on the left is named *Old*, and the one on the right is named *New*. You can think of these as being an original forecast and a revised forecast. The cells in the Old range contain values. The cells in the New range contain a formula that will calculate a random number each time you press F9 to recalculate the workbook. (The formula in those cells is `=ROUND(RAND()*50+100,0)`, which tells Excel to calculate a random number between 0 and 1, multiply it by 50, add 100, and round to the nearest whole number. Because the numbers in the New range are randomly generated, the ones you see will differ from the ones in this graphic.)

	A	B	C	D	E	F	G	H
1								
2	Old				New			
3	143	116	110		146	110	106	
4	133	136	114		146	116	137	
5	123	113	120		123	143	119	
6	103	148	129		112	134	108	
7								

The *Flow* text file contains a macro named *CompareCells*, which looks like this:

```
Sub CompareCells()
    Dim i As Integer
    Calculate
    If Range("New").Cells(i) > Range("Old").Cells(i) Then
        Range("New").Cells(i).Interior.Color = rgbLightGreen
    Else
        Range("New").Cells(i).Interior.Color = rgbLightSteelBlue
    End If
End Sub
```

The macro first executes the `Calculate` method, which calculates new values for all the cells in the `New` range. Then the macro compares only the last cell in the `New` range with the last cell in the `Old` range. If the `New` value for that one cell is greater than the `Old` value, the cell turns light green; otherwise, it turns light steel blue. The macro assigns the `Count` of cells in the range to the variable `i`, which is a simple integer.

See Also If you're not comfortable with `If` blocks, review the first half of this chapter. For more information about cell color, see the section titled "Format the Interior of a Range" in Chapter 4, "Explore Range Objects."

Now see how you can convert this macro to use a loop to compare and color all the cells in the `New` range.

1. Copy the `CompareCells` macro from the text file, and paste it into a VBA module in the `Chapter07` workbook.
2. Click in the `CompareCells` macro, and press `F8` repeatedly to step through the macro. Make sure you understand everything the original macro does.

	A	B	C	D	E	F	G	H
1								
2	Old				New			
3	143	116	110		117	141	129	
4	133	136	114		136	116	138	
5	123	113	120		103	117	109	
6	103	148	129		137	135	121	
7								

3. In the statement that assigns the `Count` to the variable, insert the word `For` in front of the variable, and then insert `1 To` after the equal sign.
4. Type `Next i` before the `End Sub` statement, and indent all the statements between `For` and `Next`.

The finished macro should look like this:

```
Sub CompareCells()
    Dim i As Integer
    Calculate
    For i = 1 To Range("New").Cells.Count
        If Range("New").Cells(i) > Range("Old").Cells(i) Then
            Range("New").Cells(i).Interior.Color = rgbLightGreen
        Else
            Range("New").Cells(i).Interior.Color = rgbLightSteelBlue
        End If
    Next i
End Sub
```

The keyword *For* works just like a simple assignment statement. It assigns a number to the variable. (The *For* statement assigns a number to an integer variable, while the *For Each* statement assigns a reference to an object variable.) The variable that holds the number is called a loop counter. You specify the start value for the loop counter (in this case, 1) and the stop value (in this case, the total number of cells in the range).

The *For* loop assigns the start value to the loop counter, executes all the statements down to the *Next* statement, adds 1 to the loop counter, and checks the loop counter against the stop value. If the loop counter is greater than the stop value, the *For* loop jumps to just past the *Next* statement. If the loop counter is less than or equal to the stop value, the *For* loop does it all again.

5. Press F8 repeatedly to watch the macro work. Step through at least two or three loops, and then press F5 to finish the macro.

	A	B	C	D	E	F	G	H
1								
2	Old				New			
3	143	116	110		121	121	142	
4	133	136	114		132	146	106	
5	123	113	120		149	145	128	
6	103	148	129		110	119	113	
7								

In many cases, using a *For Each* loop is more convenient than using a *For* loop. However, a *For* loop is a more general tool: you can always use a *For* loop to reproduce the behavior of a *For Each* loop. For example, here's how you could write the *ProtectSheets* macro without using *For Each*:

```
Sub ForProtectSheets()
    Dim mySheet As Worksheet
    Dim i As Integer
    For i = 1 to Worksheets.Count
        Set mySheet = Worksheets(i)
        mySheet.Select
        mySheet.Protect "Password", True, True, True
    Next i
End Sub
```

Troubleshooting If you run a macro that contains an infinite loop, stop the macro by pressing Ctrl+Break.

The *For* loop is a little more dangerous than a *For Each* loop because you have to be sure to get the start and stop values correct. If you have a stop value that is smaller than the start value, the loop will run forever—a condition known as an infinite loop. With a *For Each* loop, it is impossible to create an infinite loop.

Loop Indefinitely by Using a Do Loop

A For Each loop works through a collection. A For loop cycles through numbers from a starting point to an ending point. In some situations, however, neither of these options works.

For example, suppose that you want to retrieve the names of all the Excel workbooks in the current folder. Visual Basic has a function that tells you the names of files in a folder (or directory). The function is named *Dir*, after the old MS-DOS operating system command of the same name. The first time you use *Dir*, you give it an argument that tells which kind of files you want to look at. To retrieve the name of the first Excel workbook in the current directory, you use the statement `myFile = Dir("*.xlsx")`. To get the next file that matches the same pattern, you use *Dir* again, but without an argument. You must run *Dir* repeatedly because it returns only one file name at a time. When Visual Basic can't find another matching file, the *Dir* function returns an empty string.

So how do you create a macro that retrieves the names of all the Excel files in the current folder? The list of files in the directory isn't a collection, so you can't use a For Each loop. You can't use a For loop either because you don't know how many files you'll get until you're finished. Fortunately, Visual Basic has one more way of controlling a loop: a Do loop.

The ListFiles macro in the *Flow* text file retrieves the first two Excel files from the current directory and puts their names into the first two cells of the first column of the active worksheet. Here's the original macro:

```
Sub ListFiles()
    Dim myRow As Integer
    Dim myFile As String

    myRow = 1
    myFile = Dir("*.xls")
    Cells(myRow, 1) = myFile

    myRow = myRow + 1
    myFile = Dir
    Cells(myRow, 1) = myFile
End Sub
```

Aside from the variable declaration statements, this macro consists of two groups of three statements each. In each group, the macro assigns a row number to `myRow`, retrieves a file name using the *Dir* function, and then puts the file name into the appropriate cell. The first time the macro uses *Dir*, it specifies the pattern to match. The next time, the macro uses *Dir* without an argument so that it will retrieve the next matching file.

Now see how you can convert this macro to loop until it has found all the files in the folder.

1. Copy the **ListFiles** macro from the text file, and paste it into a VBA module in the *Chapter07* workbook.
2. In the *Chapter07* workbook, activate the **Files** worksheet.
3. Make sure the current folder is the one containing the practice files for this book. (Click the **Microsoft Office Button**, click **Open**, change to the correct folder, and then click **Cancel**.)
4. In the Visual Basic editor, click in the **ListFiles** macro, and press F8 repeatedly to step through the macro. (The names of the files your macro retrieves might differ from those in the graphics.) Make sure you understand the original macro.

	A	B
1	EIS.xlsx	
2	Flow.xlsx	
3		

Tip As you step through the macro, move the mouse pointer over a variable name to see the current value stored in that variable.

5. At the end of the first statement that contains a **Dir** function, insert a new line, and type **Do Until myFile = ""** (There is no space between the quotation marks.)
This statement begins the loop. You begin the loop after the first **Dir** function because you use **Dir** with an argument only once.
6. At the end of the second statement that contains a **Dir** function, insert a new line, and type **Loop**.
This statement ends the loop and sends Visual Basic back to the start of the loop to check if it's time to quit.
7. Delete the second **Cells(myRow, 1) = myFile** statement.
You don't need this statement because the loop repeats the assignment statement as many times as needed.
8. Just before the **myRow = 1** statement, insert a line, and then enter the statement **Cells.Clear**.
This ensures that the worksheet is empty in case you run the macro multiple times and some lists are shorter than others.

Tip When you use a macro to write a list onto a worksheet, make sure there are no old lists left in the worksheet. You can use **Cells.Clear** to erase the worksheet, or use **Worksheets.Add** to create a new one.

9. Indent the three statements between the **Do** and **Loop** statements.

The revised macro should look like this:

```
Sub ListFiles()
    Dim myRow As Integer
    Dim myFile As String

    Cells.Clear
    myRow = 1
    myFile = Dir("*.xlsx")
    Do Until myFile = ""
        Cells(myRow, 1) = myFile

        myRow = myRow + 1
        myFile = Dir
    Loop
End Sub
```

The `myFile = ""` expression at the end of the `Do Until` statement is a conditional expression, precisely like one you'd use with an `If` statement. The conditional expression must be something that Visual Basic can interpret as either `True` or `False`. Visual Basic simply repeats the loop over and over until the conditional expression is `True`. Note that the condition may never be true, in which case the loop will never execute. For example, if there were no `.xlsx` files in the folder, the stop condition would be true the very first time it executes.

If you want to increment a number during the loop, you must enter a statement to do so. You must always be careful to cause something to happen during the loop that will allow the loop to end. In this case, you retrieve a new file name from the `Dir` function.

10. Press F8 repeatedly to watch the macro work. Step through at least two or three loops, and then press F5 to finish the macro.

	A	B
1	EIS.xlsx	
2	Flow.xlsx	
3	Graphics.xlsx	
4	Loan.xlsx	
5	Orders.xlsx	
6	Ranges.xlsx	
7	Budget.xlsx	
8		

Troubleshooting If you run a macro that contains an infinite loop, stop the macro by pressing `Ctrl+Break`.

A Do loop is the most flexible of all the looping structures. Anything that you can do with a For loop or a For Each loop, you can do with a Do loop. If you had to be stranded on a desert island with only one loop structure, the Do loop would be the best one to have. For example, here is how you could write the ProtectSheets macro by using a Do loop.

```
Sub ProtectSheets()  
    Dim mySheet As Worksheet  
    Dim i As Integer  
    i = 1  
    Do Until i > Worksheets.Count  
        Set mySheet = Worksheets(i)  
        mySheet.Select  
        mySheet.Protect "Password", True, True, True  
        i = i + 1  
    Loop  
End Sub
```

The flexibility makes the Do loop a little more complicated than the others because you have to create and increment your own loop variable and provide your own condition for ending the loop. This makes a Do loop particularly vulnerable to becoming an infinite loop. For example, if you forgot to add the statement to retrieve a new file name, or if you had included the argument to the Dir function inside the loop (so that Dir would keep returning the first file name over and over), you'd have an infinite loop.

Tip Do loops have several useful variations. You can loop until the conditional expression is True or while the expression is True. You can put the conditional expression at the top of the loop (in the Do statement) or at the bottom of the loop (in the Loop statement). To find out more about Do loop structures, select the word Do in the macro, and then press F1.

Managing Large Loops

A loop that executes only two or three times isn't much different from a program without a loop. It runs fast, and it's easy to step through to watch how each statement works. Once you start repeating a loop hundreds or thousands of times, however, you need some additional techniques to make sure the macro works the way you want it to.

Set a Breakpoint

The *Flow* text file includes a macro named *PrintOrders*. You can think of this macro as one that your predecessor wrote just before leaving the company. Or you can think of it as one that you almost finished three months ago. In either event, you have a macro that you don't completely understand and that doesn't work quite right.

The *PrintOrders* macro is supposed to print a copy of the entire *Orders* workbook, specifically one that is sorted by product Category. You give each Category manager the section of the report that shows orders only for that one category, so you need a new page every time the Category changes. Unfortunately, the macro doesn't do what it's supposed to. You need to find and fix the problem. Here's the macro as you first receive it:

```
Sub PrintOrders()
    Dim myRow As Long
    Dim myStop As Long
    Workbooks.Open FileName:="orders.xls"
    Columns("E:E").Cut
    Columns("A:A").Insert Shift:=xlToRight
    Range("A1").CurrentRegion.Sort Key1:="Category", _
        Order1:=xlAscending, Header:=xlYes
    myStop = Range("A1").CurrentRegion.Rows.Count
    For myRow = 3 To myStop
        If Cells(myRow, 1) <> Cells(myRow + 1, 1) Then
            Cells(myRow, 1).Select
            ActiveCell.PageBreak = xlPageBreakManual
        End If
    Next myRow
    Cells(myRow, 1).Select
    ActiveSheet.PageSetup.PrintTitleRows = "$1:$1"
    ActiveSheet.PrintPreview
    ActiveWorkbook.Close SaveChanges:=False
End Sub
```

The best approach is probably to start stepping through the macro.

1. Copy the **PrintOrders** macro from the text file, and paste it into a VBA module in the *Chapter07* workbook.
2. Make sure the current folder is the one containing the practice files for this book. (Click the **Office** Button, click **Open**, change to the correct folder, and then click **Cancel**.)

3. In the Visual Basic editor, click in the **PrintOrders** macro, and then press F8 three times to jump over the variable declarations and open the *Orders* workbook.

	A	B	C	D	E	F	G	H
1	Date	State	Channel	Price	Category	Units	Net	
2	January-05	Oregon	Wholesale	High	Art	670	\$1,681.65	
3	January-05	Washington	Wholesale	High	Seattle	65	\$178.75	
4	January-05	Washington	Wholesale	High	Art	50	\$137.50	
5	January-05	Washington	Retail	High	Art	10	\$55.00	

4. Press F8 three more times.

These statements move the **Category** field over to column A and then sort the list by **Category**.

	A	B	C	D	E	F	G	H
1	Category	Date	State	Channel	Price	Units	Net	
2	Art	January-05	Oregon	Wholesale	High	670	\$1,681.65	
3	Art	January-05	Washington	Wholesale	High	50	\$137.50	
4	Art	January-05	Washington	Retail	High	10	\$55.00	
5	Art	January-05	Oregon	Wholesale	Mid	1,425	\$2,738.24	
6	Art	January-05	Washington	Wholesale	Mid	75	\$168.75	

5. Press F8 twice to assign a number to **myStop** and to start the loop. Hold the mouse pointer over **myStop** and then over **myRow** to see the values that were assigned.

The value of **myStop** is 3266, and the value of **myRow** is 3. Those values appear to be correct. The loop will execute from row 3 to row 3266.

```

myStop = Range("A1").CurrentRegion.Rows.Count
For myRow = 3 To myStop
  If myRow = 3 Then
    Cells(myRow, 1) <> Cells(myRow + 1, 1) Then
      Cells(myRow, 1).Select
      ActiveCell.PageBreak = xlPageBreakManual
    End If
  End If

```

6. Press F8 several times.

Visual Basic keeps checking whether the cell in the current row matches the cell below it. How many rows are in the **Art** category? Pressing F8 repeatedly until the macro finds the last row in the category could take a long time. But if you just press F5 to run the rest of the macro, you can't watch what happens when the condition in the **If** statement is **True**. If only there were a way to skip over all the statements until the macro moves into the **If** block.

7. Click in the gray area to the left of the statement starting with **ActiveCell**.

A dark red circle appears in the margin, and the background of the statement changes to dark red. This is a breakpoint. When you set a breakpoint, the macro stops when it reaches the breakpoint statement.

```

myStop = Range("A1").CurrentRegion.Rows.Count
For myRow = 3 To myStop
    Application.StatusBar = "Processing row " & myRow & " of " & myStop
    If Cells(myRow, 1) <> Cells(myRow + 1, 1) Then
        Cells(myRow, 1).Select
        ActiveCell.PageBreak = xlPageBreakManual
    End If
Next myRow
Cells(myRow, 1).Select
    
```

- Press F5 to continue the macro.

The macro stops at the breakpoint. When the macro reaches the breakpoint, the active cell is the first one that the If statement determined is different from the cell below it.

```

myStop = Range("A1").CurrentRegion.Rows.Count
For myRow = 3 To myStop
    Application.StatusBar = "Processing row " & myRow & " of " & myStop
    If Cells(myRow, 1) <> Cells(myRow + 1, 1) Then
        Cells(myRow, 1).Select
        ActiveCell.PageBreak = xlPageBreakManual
    End If
Next myRow
Cells(myRow, 1).Select
    
```

- Press F8 to execute the statement that assigns a manual page break.

	A	B	C	D	E	F	G	H
517	Art	October-07	Nevada	Retail	Mid	75	\$337.50	
518	Art	October-07	Oregon	Retail	Mid	50	\$225.00	
519	Art	October-07	Oregon	Wholesale	Mid	50	\$112.50	
520	Art	October-07	Utah	Retail	Mid	35	\$157.50	
521	Art	October-07	Washington	Retail	Mid	22	\$99.00	
522	Dinosaurs	January-05	Washington	Retail	Low	40	\$140.00	

The page break appears above the row, not below the row. This is a problem. The macro shouldn't set the page break on the last cell of a Category; rather, it should set the break on the first cell of a Category. The If statement should check to see whether the cell is different than the one above it.

- Change the plus sign (+) in the If statement to a minus sign (-).

The revised statement should look like this:

```
If Cells(myRow, 1) <> Cells(myRow - 1, 1) Then
```

- Click the **Reset** button, press F5, and click **Yes** to reopen the *Orders* file. Then press F8 to watch the critical statement work—properly this time—as it assigns the page break after the Art category.
- Click the red circle in the margin to turn off the breakpoint.

Setting a breakpoint is an invaluable tool for finding a problem in the middle of a long loop. In the following section, you'll learn an easy way to set a temporary breakpoint if you need to use it only once.

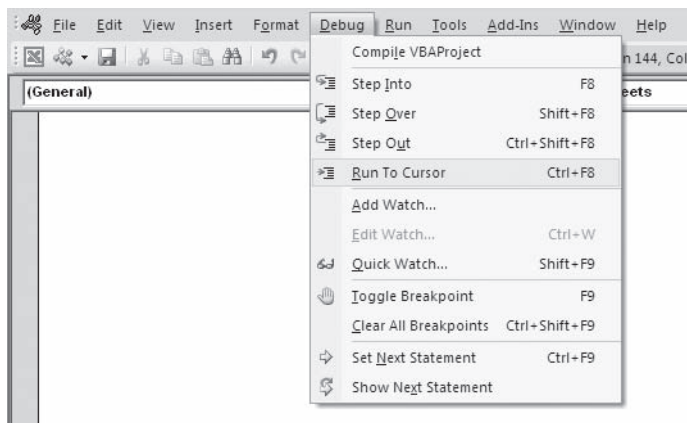
Set a Temporary Breakpoint

A breakpoint stops the macro each time the macro reaches the statement, and the breakpoint stays around until you remove it. What if you want to create a temporary breakpoint—one that you use only once? For example, suppose you're stepping through the middle of the PrintOrders macro. The code to assign a page break seems to be working properly. However, there are still some statements at the end of the macro that you'd like to step through.

1. If you're not already stepping through the macro, press F8 to start the macro.
2. Click anywhere in the **Cells(myRow, 1).Select** statement after the end of the loop to place the insertion point in that statement.

You want a breakpoint on this statement, but one that you need to use only once.

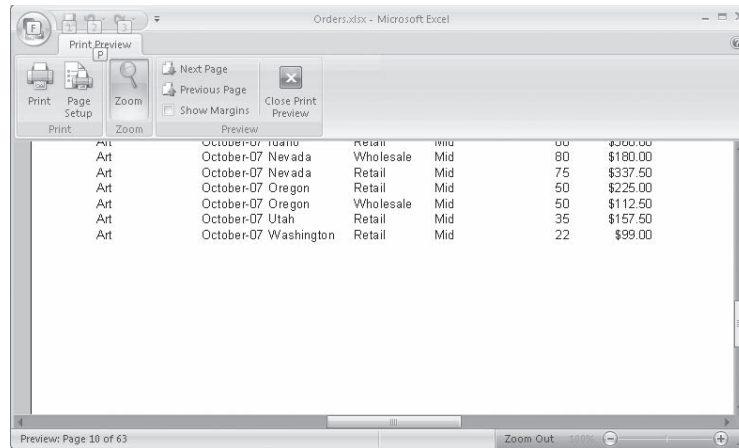
3. On the **Debug** menu, click the **Run To Cursor** command.



The macro runs through all the pages of the report and stops on the statement with the cursor.

4. Press F8 three times to scroll to the bottom of the list, set the print titles, and preview the report. Once the macro has stopped on a statement, you can continue stepping from there.
5. Review the report. Click **Next Page** repeatedly to get to page 10 to see the end of the Art category.

Troubleshooting If you don't see the end of the Art category on page 10, simply click Next Page or Previous Page to locate the correct page. Your current printer driver might have placed the end of the category on a different page.



6. Close **Print Preview**, and press F8 twice more to finish the macro.
7. Save the *Chapter07* workbook.

Turning off a breakpoint is just as easy as turning one on: just click in the left margin of the Visual Basic editor window. But if turning a breakpoint on and off is still too much work, you can create a temporary one by running to the cursor.

Show Progress in a Loop

Even if the loop in a macro is working perfectly, you might get nervous about whether something has gone wrong if the macro takes a long time to execute. The best way to feel comfortable when a long loop is running (particularly if you're wondering whether you have time to get a cup of coffee) is to show the progress of the loop.

You can show progress with any kind of loop. But a For loop lends itself particularly well to showing progress because at any point in the loop, your macro can determine both the current value of the loop counter and also what its final value will be.

1. In the **PrintOrders** macro, immediately following the **For** statement, insert this statement:

```
Application.StatusBar = "Processing row " & myRow & " of " & myStop
```

The status bar at the bottom of the Excel window usually says "Ready." The `StatusBar` property of the `Application` object allows you to make the status bar say whatever you want. The best message is one that shows progress and also gives you an idea of how long the task will take.

The statement you added creates this message when it enters the loop the first time: "Processing row 3 of 3300." By using an ampersand (&) to join together message text with the numbers in the myRow and myStop variables, you can create a useful message. Just be careful to include an extra space before and after the numbers.

2. Press F5 to run the macro. Watch the status bar to see how the macro is progressing.

527	Dinosaurs	January-05	Washington	Retail
528	Dinosaurs	February-05	Washington	Retail
529	Dinosaurs	February-05	Washington	Wholesale

Orders

Processing row 1046 of 3266

3. Close the **Print Preview** screen to let the macro finish.

The status bar indicates that the macro is still running. The status bar doesn't automatically reset when your macro ends. To return control of the status bar to Excel, you must assign it a value of False.

4. After the Next statement, insert the statement:

```
Application.StatusBar = False
```

5. Run the macro again, close the **Print Preview** screen at the appropriate time, and then look at the status bar.

It's back to normal.

13					
14					
15					

Test Compare Files

Ready

6. Save the *Chapter07* workbook.

Visual Basic provides extremely powerful tools for repeating statements in a loop. Coupled with the decisions that you can make using If blocks, these tools let you create macros that are smart and very powerful.



CLOSE the *Chapter07.xlsm* workbook.

Key Points

- Use an If structure to make a decision. Add an Else clause if you need different actions for True and False conditions.
- When you use a MsgBox, use the Buttons argument to create explicit choices. When checking the result of a MsgBox, be sure to test against the proper constant.
- When you use an InputBox, be sure to think through all the possible types of input—valid, invalid, non-existent. Create an If structure to handle all the possible conditions.
- When you simply need to loop through a collection, a For Each is the easiest option.
- When you need a counter to help you as you work through a loop—for example, to keep two objects synchronized—use a For loop.
- When you need to loop for a while or until a condition is True, use a Do loop. A Do loop is extremely flexible, but be careful that you don't create an infinite loop.
- Use permanent and temporary breakpoints when troubleshooting large macros or loops.

Index

A

- [A1 reference style](#), 119
- [Accounting format](#), 5
- [Accounting Number Format button](#), 5
- [Activate method](#), 82, 88
- [active workbook](#), referencing, 82
- [ActiveCell object](#), 53
- [ActiveCell property](#), 104–105
 - column selection with, 105
- [ActiveSheet](#), 46
- [ActiveX controls](#). *see also* [command buttons](#)
 - defined, 264
 - vs. Forms controls, 268
- [Add method](#), 71, 75
 - Name argument, empty string for, 157
 - Source value, 144
 - value returned by, 86
- [AddDataField method](#), 160
- [adjustment handles on shapes](#), 190, 191
- [animations](#), 336
- [arguments](#)
 - defined, 27
 - displaying all for method. *see* [Auto Quick Info](#)
 - for functions, adding, 240–242
 - for functions, making optional, 243–244
 - naming, 27
 - parentheses around, 218, 219
 - for PasteSpecial method, 27
 - syntax in macros, 27
- [Array function](#), 88

- [attributes](#). *see* [properties](#)
- [Auto List Members](#), 45, 74
 - for worksheets, activating, 89–90
- [Auto Quick Info](#), 45

B

- [blank cells in worksheets](#), selecting, 48–49
- [BorderAround method](#), 129
- [borders](#)
 - on ranges, adding, 127–130
 - on shapes, modifying, 192
- [Borders object](#), 129
- [breakpoints](#)
 - setting, 230
 - temporary, 232–233
- [browsing object classes](#). *see* [Object Browser](#)
- [buttons](#)
 - From Access, 143
 - Accounting Number Format, 5
 - Design Mode, 268
 - Macros, 8
 - Object Browser, 94
 - Properties, 265
 - Record Macro, 6
 - Save, 21
 - Search, 128
 - Stop Recording, 6, 7, 115
 - Undo, 18
 - Use Relative References, 59
 - View Code, 267

C

Calculate method, 71

cancel buttons on user forms, 314–315

canceling events, 279–280

cell styles

built-in, 132

modifying, 133

cells

Currency format, applying, 5–7

formatting, with built-in tool, 5

formatting, with macro, 6–7

Cells method, 126

Cells property, 98–100

selecting all cells with, 98

selecting specific cells with, 99

cells, table

merging and centering. *see* Merge And

Center button

referencing, 152

cells, worksheet

formulas, inserting in all selected, 49

gradient fills. *see* gradient fills

properties for, 122

referencing by number, 99

referencing by row and column,
99–100

referencing, difference in Excel
2007, 100

relative references vs. absolute
references, 58–60

selecting all, 101

selecting blank, 48–49

style, assigning to. *see* cell styles

Chart object, 203

chart objects

accessing without selecting, 203

axes, fixed, 204

creating, 201–202

defined, 201

formatting, 207

gradient fills, applying to elements
of, 206

naming, 203

synchronizing, 203–205

type, selecting, 202

ChDir statement, 43

check boxes

adding to forms, 309

captioning, 309

defined, 309

event handlers for, 310

implementing, 327–328

sizing, 309

clearing formatting from worksheets, 134

Click event, 310

CLng function, 241

Close method, 78

closing

databases, 60

workbooks, without saving changes, 83

code name for worksheets, 183

collections

adding items to. *see* Add method

Array function unavailable for, 89

ColorStops, 180

CustomViews, 320

defined, 68

GradientStops, 184

items in, vs. instances, 81

looping through, 220–222

naming items in, 81–82

properties of, 70

referencing specific items in, 79–81,
81–82, 197

subcollections for. *see* subcollections

colon-equal signs in macros, 27

colors

- setting, 131–135
- specifying, 131

ColorStops collection, 180**columns, PivotTable**

- autofitting, 166
- fill color, 168
- width, standardizing, 163–164

Columns property, 101–103

- selecting all cells with, 101

columns, table

- captioning, 150
- inserting, 146

columns, worksheet

- hiding dynamically, 321–325
- inserting, 52
- referencing by letter, 102
- referencing by number, 101
- referencing by range, 102
- width, specifying, 177

combo boxes

- column widths, setting, 296–297
- event handlers, creating, 297–299
- inserting, 293–294
- populating lists with, 295–296
- styles, 294

command buttons

- aligning, 265
- appearance of, 264
- as cancel buttons, 314–315
- captioning, 266
- copying, 314
- default on form, 314
- design mode vs. run mode, 267–268
- event handlers, adding, 315
- events recognized by, 270
- inserting, 265
- macros, linking to, 267–268

- mouse movements, responding to, 270–271

- naming, 265–266

- properties, displaying, 265–266

- snapping to grid, 265

- TakeFocusOnClick property, 266

- in user forms, adding, 314–316

CommandButton button, 314**comments, 11****compiler errors, 246****Complete Word command, 104****conditional expressions**

- defined, 211
- in Do statements, 227
- for input boxes, 214–215
- with MsgBox function, 219
- multiple conditions in, 212–213

conditional formats, 182**confirmation dialog box, creating. *see* MsgBox function****controls. *see also* check boxes; option buttons; scroll bars; spin buttons; text boxes**

- copying, 289
- initializing, 314
- option buttons, checking for, 326
- tab order, setting, 316–317

Copy method, 87**copying**

- controls, 289
- worksheets, macro for, 87

Count property for Workbooks object, 77–78**CreateNames method, 125****CreatePivotTable method, 155****Currency format**

- vs. Accounting format, 5
- applying to cells, 5–7
- customizing, 6–7

current region

- defined, 48
- selecting, 48, 49

CurrentRegion property, 104

- with multicell range as starting point, 106

custom dialog boxes. *see* user forms**CustomView object, 320****CustomViews collection, 320****D****data sources, 138****data types, converting between. *see* Type Conversion Functions****databases**

- closing, 60
- saving changes when closing, 60–61

DataBodyRange property, 149**Date function, 312****dates**

- filling range with, 52
- finding, 321–322
- formatting as start of month, 312–313
- hiding columns preceding, 321–325
- prompting for, 53–54

declaring variables, 123, 180**Delete method**

- confirmation prompt, turning off, 62
- value returned by, 85

deleting

- digital IDs, 34
- macros, 60
- worksheet rows, 41
- worksheets, 61–62
- worksheets, macro for, 85

design mode, 267–268

- switching to, 289

Design Mode button, 268**Developer tab (Ribbon)**

- components of, 264
- displaying, 8, 264

dialog boxes

- custom. *see* user forms
- Digital Signature, 32
- Get A Digital ID, 31
- Go To Special, 48
- Macro Options, 8
- Microsoft Office Trusted Location, 29
- Modify Button, 23
- Open, 46
- Paste Special, 24
- Tab Order, 316
- Trust Center, 29

digital IDs

- creating, 31–32
- deleting, 34

Digital Signature dialog box, 32**Dim statement, 89, 90****Dir function, 225****DisplayAlerts property, 62****Do loops, 225–228**

- for error resolution, 254–255
- flexibility of, 228

docking

- Immediate window, 76
- task panes, 76

down payments, calculating. *see* loan payment calculator**drawing objects. *see* shapes****drop shadows. *see* font shadows on shapes****E****editing macros, 9–10****Else statement, 212–213. *see also* If blocks**

Elseif statement, 213. *see also* If blocks

embedded charts. *see* chart objects

End Sub statements, 11

enterprise information system (EIS), 335–338

enumerated lists, 128

Err object

- defined, 252
- Description property, 258
- error checking with, 253

error checking, 252, 253

error handlers, 257

- when to use, 258–259

error messages

- clearing, 248
- creating, 258

error trapping, 256–259

ErrorHandler statement, 256–257

errors

- compiler, 246
- logic, 246
- run-time, 246
- run-time, ignoring, 249–251
- run-time, trapping, 256–259
- syntax, 245

event handler procedures

- arguments, 270
- defined, 262
- storage of, 272

event handlers

- for check boxes, 310
- for command buttons, 315
- for controls, creating, 297–299
- creating, 269–270
- defined, 267
- for workbooks, 276–277
- for workbooks and worksheets, setting precedence for, 277–279

events

- canceling, 279–280
- for user forms, default, 311

Excel methods, 244

Excel tables. *see* tables

Exit Sub statement, 219

F

file extensions for workbooks, 12

FillFormat object, 183

- referencing, 206

FillLabels macro, stepping through, 50–51

filtering tables, 148

finding dates, 321–322

Finished folder, xv

floating

- Immediate window, 76
- task panes, 76

font shadows on shapes, 194

fonts in shapes, 192–193

For Each blocks, 220–222

- vs. For loops, 224–225

For loops, 222–224

- vs. For Each blocks, 224–225

formatting text in shapes, 192–193

Forms controls

- vs. ActiveX controls, 268
- selecting, 268

forms, user. *see* user forms

formula bar, worksheet. *see* window elements in worksheets

Formula property, 122

FormulaR1C1 property, 122

formulas

- A1 reference style, 119
- absolute references in, 117–118
- dollar signs in, 117–118

formulas (continued)

- for filling grids, 123–127
- inserting in all selected cells, 49
- R1C1 reference style, 51, 119–120
- R1C1 reference style syntax, 120
- relative references in, 116–117
- simplifying, 150
- values, converting to with macro, 26–27

From Access button, 143**function declaration statement, 240****functions**

- arguments, adding, 240–242
- arguments, making optional, 243–244
- Array, 88
- CLng, 241
- creating, 239–240
- custom, from macros, 244–245
- data type conversion with. *see* Type Conversion Functions
- Date, 312
- defined, 238
- Dir, 225
- InputBox, 53
- MsgBox, 217–219
- recalculating when any cell changes. *see* volatile functions
- Rnd, 239, 241
- testing, from user forms, 313
- volatile. *see* volatile functions

G**Get A Digital ID dialog box, 31****<globals> object class, 95****Go To Special dialog box, 48****gradient fills**

- adding to cells, 178–182
- color stop positioning in, 181

- multiple colors in, 180

GradientStops collection, 184**GradientStops property, 180, 183****graphical objects. *see also* shapes**

- defined, 176

graphical user interface

- defined, 284

gridlines, worksheet. *see* window elements in worksheets**grids**

- defining space in, 124–125
- formulas for filling, 123–127

grouping shapes, 196–200**H****hardware requirements for 2007**

- Microsoft Office, xvii

headings, sidebar. *see* sidebar

- headings

headings, worksheet. *see* window elements in worksheets**Help**

- with book and CD, xxi
- member list for objects, 178

hidden objects, 187**hiding**

- columns, dynamically, 321–325
- rows, 318

I**If blocks, 211. *see also* conditional expressions**

- creating, 212
- Else statements in, 212–213
- nested, 216

Immediate window, 73

- docking and floating, 76
- moving to bottom of, 79
- moving to top of, 81
- multiple statements on single line
 - in, 221
- opening, 73
- question mark in, 77
- statement execution in, 75

indenting macro statements, 28, 211**infinite loops, 224****InputBox function, 53**

- Cancel button behavior, 214–215
- validity of user input, checking, 215–216

instances, 71

- vs. items in collections, 81

interest rate, calculating. *see* loan payment calculator**Intersect property, 109****Item property, 71****items, 81**

K

keyboard shortcuts

- built-in, 9
- macros, assigning to, 8–9, 16

L

labels, 256–257**line breaks in macro statements, 28, 43****list boxes**

- inserting, 293–294
- properties, changing, 294

ListObjects. *see also* tables

- object model for, 151

lists, enumerated, 128**loan payment calculator**

- creating, 285–286
- down payment, restricting to valid values, 289–291
- implementing, 286–287
- interest rate, restricting to valid values, 291–292
- protecting, 297
- years, restricting to valid range, 288–289

Locals window, 76**logic errors, 246****Long integers, 241****loop counters, 224****loops**

- analyzing, 229–230
- breakpoints. *see* breakpoints
- Do. *see* Do loops
- ending, 227
- For. *see* For loops
- For Each. *see* For Each blocks
- infinite, stopping, 224
- progress display for, 233–234

M

macro-enabled workbooks. *see also* macros; workbooks

- vs. macro-free workbooks, 12
- saving workbooks as, 12
- trusting, 28–35

Macro Options dialog box, 8**macros**

- for appending data to master lists, 55–56
- body, defined, 11
- breakpoints. *see* breakpoints
- colon-equal signs in, 27

macros (continued)

- command buttons for. *see* command buttons
 - comments in. *see* comments
 - cursor, stopping on, 232
 - for date insertion, 52
 - for date prompting, 53–54
 - defined, 2
 - deleting, 60
 - description, changing, 8
 - editing, 9–10
 - enabling in workbooks, 13
 - errors. *see* errors
 - finding, in Procedure list, 59–60
 - formatting cells with, 6–7
 - full path, when to use, 139
 - indenting lines in, 28
 - looping. *see* loops
 - methods. *see* methods
 - modules saved in, 16
 - naming, 6
 - objects in. *see* objects
 - vs. procedures, 250, 267
 - properties. *see* properties
 - Quick Access Toolbar, adding to, 22–23
 - recording, 6
 - recording, deleting extra lines after, 17–18, 43
 - relative references vs. absolute references, 58–60
 - for removing window elements from worksheets, 18–19
 - running, 8, 19
 - running multiple times. *see* loops
 - for running other macros, 63–64
 - shortcut keys, assigning to, 8–9, 16
 - for side-bar heading creation, 14–15
 - statements, line breaking, 28
 - status bar, updating with loop progress, 233–234
 - stepping through, 20, 42–45
 - stepping through, editing statements during, 44
 - stopping, 224
 - for text file opening, 39–41
 - user forms for. *see* user forms
 - user input cancellation, allowing, 214–215
 - vs. VBA, 2–4
 - workbook contained in, returning, 140
 - for workbook creation, 73–75
 - for worksheet copying, 87
 - for worksheet deletion, 61–62, 85
 - for worksheet selection, 88
- Macros button, 8**
- master lists, appending data to, 55–56**
- members, 71**
- memory caches for PivotTable reports, 155**
- Merge And Center button, 14**
- methods. *see also* specific methods**
- Activate, 82, 88
 - Add, 71, 75
 - AddDataField, 160
 - arguments. *see* arguments
 - BorderAround, 129
 - Calculate, 71
 - Cells, 126
 - Close, 78
 - Copy, 87
 - CreateNames, 125
 - CreatePivotTable, 155
 - defined, 24
 - Delete, 62, 85
 - displaying all for object. *see* Auto List Members
 - Excel, 244

- object classes and, 70
- PasteSpecial, arguments for, 27
- vs. properties, 25, 27, 28, 72
- vs. properties, in recorded macros, 72
- property changes by, 72
- Protect, 299
- Range, 126
- Select, 88
- Show, 332
- ShowAllData, 152–153
- SpecialCells, 132
- syntax in macros, 27
- values returned by, 85–86
- xldataField, 160
- Microsoft Office Fluent user interface, xiii**
- Microsoft Office Trusted Location dialog box, 29**
- Microsoft Visual Basic for Applications (VBA). *see* VBA**
- modal forms, 332**
- Modify Button dialog box, 23**
- module-level variables, 278**
- modules, saved macros in, 16**
- months, setting dates as first day of, 312–313**
- MouseMove event, 270–271**
- MsgBox function**
 - arguments for, 218
 - conditional expressions and, 219
 - defined, 217
 - return values, 219

N

naming

- chart objects, 203
- command buttons, 265–266

- items in collections, 81–82
- macros, 6
- ranges, 285, 293
- Shape objects, 185
- user forms, 305–306
- worksheets, macro for, 85
- .NET Framework and VBA, 4**
- Not keyword, 20–21**
- numbers, Long. *see* Long integers**

O

Object Browser, 76

- components of, 95
- hidden members, displaying, 187

Object Browser button, 94

object classes

- browsing, with Object Browser. *see* Object Browser
- defined, 69
- instances, 71
- members, 71
- methods of, 70. *see also* methods
- properties of, 70. *see also* properties

object model

- defined, 3
- languages communicated with, 4
- VBA interface with, 3

objects

- ActiveCell, 53
- Borders, 129
- Chart, 203
- CustomView, 320
- in collections. *see* collections
- defined, 3, 67, 68–69
- Err, 252–253, 258
- FillFormat, 183, 206
- graphical. *see* graphical objects

objects (continued)

- hidden, 187
- members, viewing list of, 178
- PivotCache, 156
- properties. *see* properties
- Range, 107–111, 132
- repeated, consolidating, 21–22
- returned by properties, finding, 183
- selecting multiple, 200
- Selection, 53
- Shape, 188
- ShapeRange, 188
- Shapes, 188
- specifying, 11
- syntax in macros, 26–27
- for user forms, 311
- UserForm, 311
- Workbook, 320
- WorkbookConnection, 157
- Workbooks, 73, 77–78, 83
- Worksheet, 97
- Worksheets, 84–85, 97
- Office Fluent user interface, xiii**
- Offset property, 109–111**
- On Error Resume Next statement, 249–251**
- Open dialog box, displaying with macro, 46**
- OpenText statement, 43**
- option buttons**
 - adding to forms, 307, 325–327
 - aligning, 308
 - captioning, 307
 - defined, 306
- Option Explicit statement, 246**
- optional arguments, 243–244. *see also* arguments**

P

- Paste Special dialog box, 24**
- PasteSpecial method, 27**
- Pattern property, 179**
- payments, calculating for loan. *see* loan payment calculator**
- PivotCache object, 156**
- PivotTable reports**
 - columns, autofitting, 166
 - columns, coloring, 168
 - columns, standardizing width, 163–164
 - creating, from external sources, 156–158
 - creating, from internal sources, 153–156
 - data fields, referencing, 161
 - data fields, renaming, 158–159
 - dates, formatting, 159
 - defined, 153
 - headers, turning off, 164
 - items with no data, displaying, 162–163
 - layout, customizing, 162–167
 - memory caches, connecting to same, 155
 - memory caches, loading for, 155
 - structuring, 158–162
 - styles, copying, 168
 - styles, customizing, 167–169
 - text fields, moving outside Row Labels area, 159
- practice files on CD, xv–xvi**
 - browsing to, xix
 - installing, xviii–xix
 - uninstalling, xx

printing reports, 329–331

private procedures, 250

procedures

vs. macros, 250, 267

private, 250

Project Explorer window, 76

properties

actions and, 72

changing, 83

changing multiple at once, 13–18

defined, 11, 69

deleting unnecessary inserted by
macro recorder, 17–18

displaying all for object. *see* Auto List
Members

Help topics for, displaying, 177

vs. methods, 25, 27, 28, 72

vs. methods, in recorded macros, 72

object for, recognizing, 17

objects returned by, finding, 183

read-only, 78, 83

read-write, 83

referencing multiple, 17

state, viewing current, 20

syntax in macros, 17, 18

toggleing values of, 20–21

unsupported, 11

Properties button, 265

Properties window, 76

Protect method, 299

protecting worksheets, 297, 299–300

public variables, 278

publishers, setting yourself as, 31–34

Q

Quick Access Toolbar, adding macros
to, 22–23

R

R1C1 reference style, 51, 119–120

syntax, 120

random number function. *see* Rnd
function

Range method, 126

Range object

Intersect property, 109

Offset property, 109–111

properties of, 107

properties, referencing, 109

Resize property, 109, 111

Rows property, 108–109

SpecialCells method, 132

Range property, 94–97

arguments for, 95

ranges

borders, adding, 127–130

cells in, referencing by attribute, 132

as collections of cells. *see* Cells
property

as collections of rows and columns. *see*
Columns property; Rows property

constant values for, 121

counting cells in, 97

filling with formulas, 123–127

names of, 96

naming, 285, 293

offset, selecting. *see* Offset property

referencing, 108–109

referring to, with addresses, 94–97

relative, referencing, 109–113

resizing when selecting. *see* Resize
property

selecting, 96–97

ranges (continued)

Value property. *see* Value property variables as, 115

read-only properties, 78, 83

read-write properties, 83

Record Macro button, 6

recording macros, 6

deleting extra lines after, 17–18, 43

relative references vs. absolute

references, 58–60

Rectangle class, 185, 187

regions, referencing. *see*

CurrentRegion property

Relative References option, 58–60

reports, printing, 329–331

Resize property, 109, 111

Ribbon

customizing with VBA, 281

Developer tab. *see* Developer tab (Ribbon)

Microsoft Office Fluent user interface, xiii

Rnd function, 238, 241

Rows property, 101–103

rows, worksheet

height, specifying, 176–177

hiding, 318

referencing by number, 102

referencing by range, 103

run mode, 267–268

Run Sub/UserForm button, 305

run-time errors, 246

ignoring, 249–251

trapping, 256–259

running macros, 8, 19

multiple times. *see* loops

from Quick Access Toolbar, 22–23

with shortcut keys, 8–9

step by step, 20, 42–45

step by step, editing statements

during, 44

by using macros, 63–64

running to the cursor, 232

S

Save button, 21

SaveChanges argument, 60

Saved property, 83

saving

database changes when closing, 60–61

workbooks, 21

scroll bars

inserting, 291

moving, 291

properties, changing, 292

Search button, 128

Select groups

replacing with With structures, 116

simplifying, 115–116

Select method, 88

row selection with, 107

Select Objects mode, 200

Select statements, deleting

unnecessary, 113–114

selecting

current region, 48, 49

Forms controls, 268

lines in VBA code, 17

objects, multiple, 200

ranges, 96–97

worksheets, all cells on, 98

worksheets, macro for, 88

worksheets, multiple, 62

Selection object, 53

Selection property, 105

Shape objects

- Auto List Help for, 182–183
- defined, 188
- FillFormats, 183
- GradientStops collection, adding to, 184
- GroupItems property, 197
- OnAction property, 199, 200

shape ranges

- defined, 188
- for multiple shapes, 188

ShapeRange objects, 188**ShapeRange property, 187****shapes. *see also* graphical objects**

- adjustment handles, 190, 191
- borders, modifying, 192
- changing, 184
- charts as. *see* chart objects
- in collections, referencing single, 197
- creating, 182–183, 189–190
- fill color, changing, 192
- font shadows, adding, 194
- formatting, in groups, 198
- gradients, adding, 183
- grouping, 196–200
- linking to macros, 199
- multiple, modifying, 188
- naming, 185
- snapping to grid, 190
- text, adding, 192
- text, formatting, 192–193
- 3-D formatting, 194–195
- transparency, changing, 192
- ungrouping, 199, 200

Shapes object, 188

- defined, 188

shortcut keys

- built-in, 9

- macros, assigning to, 8–9, 16

Show method, 332**ShowAllData method, 152–153****sidebar headings**

- defined, 14
- macro for creating, 14–15

snapping shapes to grid, 190**SpecialCells method, 132****spelling, checking in variables, 246****spin buttons**

- copying, 289
- with fractional values, 289–290
- inserting, 288
- snapping to grid, 288
- valid entries, entering, 288

statements. *see also* specific statements

- defined, 11
- line breaking, 28, 43

status bar, displaying loop progress on, 233–234**Stop Recording button, 6, 7, 115****Sub statements, 11****subcollections, 88****subroutines**

- editing, 64–65
- generalizing, 251–252

syntax errors, 245**T****Tab Order dialog box, 316****tables. *see also* ListObjects**

- cell references, 152
- columns, captioning, 150
- tables (continued)

- columns, inserting, 146
- creating, from external sources, 142–145
- creating, from internal sources, 140–142
- data sources. *see* data sources
- filter states, different for multiple, 152–153
- filtering, by top 10, 148
- header row, moving to, 152
- Total row, adding, 151

task panes, 76

text boxes

- adding to user forms, 310
- errors, checking for, 328–329
- initializing, 311–314

text files, opening with macro, 39–41

Text Import Wizard, 40

Text property, 122

ThisWorkbook property, 140

3-D formatting of shapes, 194–195

tiling windows, 15

Toolbox

- displaying, 307
- for user forms, 306

transparency in shapes, 192

trapping errors, 256–259

Trust Center dialog box, 29

trusting macro-enabled worksheets, 28–35

2007 Microsoft Office components, xvi–xvii

Type Conversion Functions, 241

U

Undo button, 18

ungrouping shapes, 199, 200

Up Arrow Callout shapes, 191

Use Relative References button, 59

user forms

- cancel buttons, 314–315
- captioning, 305–306, 307
- check boxes on. *see* check boxes
- columns, hiding dynamically, 321–325
- command buttons, adding, 314–316
- controls. *see* controls
- creating, 305–306
- displaying, 305
- events, default, 311
- frames, adding, 307
- functionality, 304
- functions, testing, 313
- implementation, 304
- interface, designing, 304
- launching, 331–332
- modal, 332
- object name, 311
- option buttons on. *see* option buttons
- tab order, setting, 316–317
- text boxes, adding, 310
- Toolbox for, 306

user input

- cancelling, allowing for, 214–215
- requiring. *see* MsgBox function
- validity, checking, 215–216

user interface, Microsoft Office

Fluent, xiii

UserForm object, 311

V

Value property, 120–121, 122

- vs. Value2 property, 122

variables, 180

- with assigned values, 238
- declaring, 123, 180
- Empty value, 323

module-level, 278
 naming, 143
 Nothing value, 323
 public, 278
 as ranges, 107, 115
 spelling errors, checking for, 246

VBA

apostrophes beginning lines in. *see* comments
 green lines in. *see* comments
 vs. macros, 2–4
 .NET Framework and, 4
 object model. *see* object model
 selecting lines in, 17
 statements. *see* statements

vbCrLf constant, 258

View Code button, 267

views

custom, creating, 317–320
 switching between, 320–321

viruses, prevention of, 12, 28

Visual Basic editor

Immediate window, 73
 opening, 73
 windows in, 76–77

volatile functions

defined, 242
 recalculating, 242
 when to use, 243

W

Watch window, 76

window elements in worksheets, removing, 18–19

windows, rearranging, 15

With structures, 17

objects, consolidating with. *see* objects
 replacing Select groups with, 116

Workbook object, 320

WorkbookConnection object, 157

workbooks

active, referencing, 82
 architecture of, 67
 closing, macro for, 78
 closing without saving changes, 83
 counting, macro for, 77–78
 creating, 143, 144
 creating, macro for, 73–75
 digital signatures, adding, 32
 event handlers for, 276–277
 events available for, 276
 file extensions, 12
 macro-free vs. macro-enabled, 12
 macros, enabling, 13
 master list. *see* master lists
 moving worksheets between, 140
 referring to, 69, 81–82
 saving, 21
 specifying, in macros, 79–81
 trusting macro-enabled, 28–35
 viruses in. *see* viruses
 windows for, 76

Workbooks object

Add method, 73
 Close method, 78
 Count property, 77–78
 Saved property, 83

Worksheet object, 97

worksheets

activating from selection, 88
 adding, macro for, 84
 Auto List Members, activating, 89–90
 blank cells, selecting, 48–49

worksheets (continued)

code name, identifying, 183
colors in. *see* colors
columns. *see* columns, worksheet
constants, displaying only, 318
copying, macro for, 87
creating new files from, 138–140
current region. *see* current region
deleting, 61–62
deleting, macro for, 85
events available for, 275
formatting, clearing, 134
formulas, inserting in all selected cells, 49
moving to different workbooks, 140
name, displaying, 84
naming, macro for, 85
naming of identical copies, 41
protecting, 297, 299–300
referencing cells in, different in Excel 2007, 100
relative references vs. absolute references, 58–60
rows, deleting, 41

selecting all cells on, 98, 101
selecting, macro for, 88
selecting multiple, 62
views. *see* views
window elements, removing with macro, 18–19
Zoom level, changing, 262–263

Worksheets object

Add method, 84
Name property, 85
vs. Worksheet object, 97

X

[xlDataField method, 160](#)

Z

[Zoom level in worksheets, 262–263](#)