

SECOND EDITION

Join the discussion @ p2p.wrox.com



Excel VBA



24-Hour Trainer

Tom Urtis

EXCEL® VBA 24-HOUR TRAINER

INTRODUCTION xxvii

► **PART I: UNDERSTANDING THE BASICS**

LESSON 1:	Introducing VBA.....	3
LESSON 2:	Getting Started with Macros	11
LESSON 3:	Introducing the Visual Basic Editor	25
LESSON 4:	Working in the VBE	33

► **PART II: DIVING DEEPER INTO VBA**

LESSON 5:	Object-Oriented Programming: An Overview	49
LESSON 6:	Variables, Data Types, and Constants	55
LESSON 7:	Understanding Objects and Collections	67
LESSON 8:	Working with Ranges	75
LESSON 9:	Making Decisions with VBA	85

► **PART III: BEYOND THE MACRO RECORDER:
WRITING YOUR OWN CODE**

LESSON 10:	Repeating Actions with Loops.....	101
LESSON 11:	Programming Formulas	113
LESSON 12:	Working with Arrays.....	127
LESSON 13:	Automating Procedures with Worksheet Events.....	137
LESSON 14:	Automating Procedures with Workbook Events	149
LESSON 15:	Handling Duplicate Items and Records	161
LESSON 16:	Using Embedded Controls.....	181
LESSON 17:	Programming Charts	199
LESSON 18:	Programming PivotTables and PivotCharts	213
LESSON 19:	User-Defined Functions	237
LESSON 20:	Debugging Your Code	251

► PART IV: ADVANCED PROGRAMMING TECHNIQUES

LESSON 21:	Creating UserForms	271
LESSON 22:	UserForm Controls and Their Functions	285
LESSON 23:	Advanced UserForms	305
LESSON 24:	Class Modules	321
LESSON 25:	Add-Ins	335
LESSON 26:	Managing External Data	353
LESSON 27:	Data Access with ActiveX Data Objects	365
LESSON 28:	Impressing Your Boss (or at Least Your Friends)	373

► PART V: INTERACTING WITH OTHER OFFICE APPLICATIONS

LESSON 29:	Overview of Office Automation from Excel	391
LESSON 30:	Working with Word from Excel	399
LESSON 31:	Working with Outlook from Excel	409
LESSON 32:	Working with Access from Excel	419
LESSON 33:	Working with PowerPoint from Excel	431
INDEX		441

Excel® VBA 24-Hour Trainer

Excel® VBA 24-Hour Trainer

Second Edition

Tom Urtis



A Wiley Brand

Excel® VBA 24-Hour Trainer, Second Edition

Published by

John Wiley & Sons, Inc.
10475 Crosspoint Boulevard
Indianapolis, IN 46256
www.wiley.com

Copyright © 2015 by John Wiley & Sons, Inc., Indianapolis, Indiana

Published simultaneously in Canada

ISBN: 978-1-118-99137-4

ISBN: 978-1-118-99140-4 (ebk)

ISBN: 978-1-118-99141-1 (ebk)

Manufactured in the United States of America

10 9 8 7 6 5 4 3 2 1

No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except as permitted under Sections 107 or 108 of the 1976 United States Copyright Act, without either the prior written permission of the Publisher, or authorization through payment of the appropriate per-copy fee to the Copyright Clearance Center, 222 Rosewood Drive, Danvers, MA 01923, (978) 750-8400, fax (978) 646-8600. Requests to the Publisher for permission should be addressed to the Permissions Department, John Wiley & Sons, Inc., 111 River Street, Hoboken, NJ 07030, (201) 748-6011, fax (201) 748-6008, or online at <http://www.wiley.com/go/permissions>.

Limit of Liability/Disclaimer of Warranty: The publisher and the author make no representations or warranties with respect to the accuracy or completeness of the contents of this work and specifically disclaim all warranties, including without limitation warranties of fitness for a particular purpose. No warranty may be created or extended by sales or promotional materials. The advice and strategies contained herein may not be suitable for every situation. This work is sold with the understanding that the publisher is not engaged in rendering legal, accounting, or other professional services. If professional assistance is required, the services of a competent professional person should be sought. Neither the publisher nor the author shall be liable for damages arising herefrom. The fact that an organization or Web site is referred to in this work as a citation and/or a potential source of further information does not mean that the author or the publisher endorses the information the organization or Web site may provide or recommendations it may make. Further, readers should be aware that Internet Web sites listed in this work may have changed or disappeared between when this work was written and when it is read.

For general information on our other products and services please contact our Customer Care Department within the United States at (877) 762-2974, outside the United States at (317) 572-3993 or fax (317) 572-4002.

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

Library of Congress Control Number: 2015930536

Trademarks: Wiley, the Wiley logo, Wrox, the Wrox logo, Programmer to Programmer, and related trade dress are trademarks or registered trademarks of John Wiley & Sons, Inc. and/or its affiliates, in the United States and other countries, and may not be used without written permission. Excel is a registered trademark of Microsoft Corporation. All other trademarks are the property of their respective owners. John Wiley & Sons, Inc., is not associated with any product or vendor mentioned in this book.

To my father, Bill Urtis.

ABOUT THE AUTHOR



TOM URTIS is a Microsoft Excel MVP who has been using Excel since 1994. Tom owns Atlas Programming Management (www.atlaspm.com), a Microsoft Office solutions company specializing in Excel programming, development, testing, and training for an international clientele. As an Excel trainer, Tom created the Excel Aptitude Test (XAT®, <http://xat.atlaspm.com>), an innovative test that measures knowledge and aptitude of Excel for individuals and businesses.

Tom has co-authored *Don't Fear The Spreadsheet* (Holy Macro! Books, 2012) and *Holy Macro, It's 2500 Excel VBA Examples* (Holy Macro! Books, 2005), and he has served as technical editor and consultant for other Excel books and training material. Tom actively contributes to the Excel community through his blog, in forums, and with his daily Excel tips and examples on social media.

Tom is a graduate of Michigan State University. He has lived in the San Francisco Bay Area since 1983, where he enjoys the outdoor life that California offers. Tom is an avid fan of college and professional sports, and a collector of rare sports memorabilia. Tom can be reached by e-mail at tom@atlaspm.com.

ABOUT THE TECHNICAL EDITOR

Mike Alexander is a Microsoft Certified Application Developer (MCAD) and author of more than a dozen books on advanced business analysis with Microsoft Access and Excel. He has more than 16 years experience consulting and developing Office solutions. Michael has been named a Microsoft MVP for his ongoing contributions to the Excel community.

CREDITS

EXECUTIVE EDITOR
Carol Long

PROJECT EDITOR
Charlotte Kughen

TECHNICAL EDITOR
Michael Alexander

PRODUCTION EDITOR
Christine O'Connor

COPY EDITOR
Kim Cofer

MANAGER OF CONTENT DEVELOPMENT AND ASSEMBLY
Mary Beth Wakefield

MARKETING DIRECTOR
David Mayhew

MARKETING MANAGER
Carrie Sherrill

PROFESSIONAL TECHNOLOGY & STRATEGY DIRECTOR
Barry Pruett

BUSINESS MANAGER
Amy Knies

ASSOCIATE PUBLISHER
Jim Minatel

PROJECT COORDINATOR, COVER
Brent Savage

PROOFREADER
Josh Chase, Word One New York

INDEXER
Ted Laux

COVER DESIGNER
Wiley

COVER IMAGE
Wiley

ACKNOWLEDGMENTS

THE PRODUCTION OF THIS BOOK WAS made possible by the combined efforts of highly talented people, starting with the entire Wiley Publishing team, all of whom are a pleasure to work with. Thanks to Carol Long, the executive editor who got the project approved, and kept the process moving from start to finish. Thanks to Technical Editor Mike Alexander, who introduced me to Wiley Publishing in 2010 when I wrote the first edition to this book. Thanks to Charlotte Kughen, the project editor; to Kim Cofer, the copy editor; and to Christine O'Connor, the production editor.

Thank you to my family and friends for your understanding and support of my book-writing schedule, and of my everyday drive for working with Excel and teaching it to others. Many thanks to the Excel development team at Microsoft Corporation for improving Excel with each new release of Office, while considering suggestions from Excel users. A special thanks to the global Excel community. You've shown me creative ways to use Excel over the years, and taught me how to explain technical concepts to beginning Excel users.

Finally, I want to thank you for buying this book. Please tell us what you think about it, including what you liked so we keep doing it, or what you think can be improved. After all, this is your book.

CONTENTS

INTRODUCTION

xxvii

PART I: UNDERSTANDING THE BASICS

LESSON 1: INTRODUCING VBA	3
What Is VBA?	3
A Brief History of VBA	4
What VBA Can Do for You	5
Automating a Recurring Task	5
Automating a Repetitive Task	5
Running a Macro Automatically if Another Action Takes Place	5
Creating Your Own Worksheet Functions	7
Simplifying the Workbook's Look and Feel for Other Users	7
Controlling Other Office Applications from Excel	7
Liabilities of VBA	8
Try It	9
LESSON 2: GETTING STARTED WITH MACROS	11
Composing Your First Macro	11
Accessing the VBA Environment	11
Using the Macro Recorder	16
Running a Macro	21
The Macro Dialog Box	21
Shortcut Key	22
Try It	22
Lesson Requirements	22
Hints	22
Step-by-Step	23
LESSON 3: INTRODUCING THE VISUAL BASIC EDITOR	25
What Is the VBE?	25
How to Get Into the VBE	25
Understanding the VBE	26
Understanding Modules	28

Using the Object Browser	28
Exiting the VBE	30
Try It	30
LESSON 4: WORKING IN THE VBE	33
Toolbars in the VBE	33
Macros and Modules	33
Locating Your Macros	33
Understanding the Code	36
Editing a Macro with Comments and Improvements to the Code	37
Deleting a Macro	39
Inserting a Module	39
Renaming a Module	41
Deleting a Module	42
Locking and Protecting the VBE	43
Try It	44
Lesson Requirements	44
Hints	44
Step-by-Step	45
PART II: DIVING DEEPER INTO VBA	
LESSON 5: OBJECT-ORIENTED PROGRAMMING: AN OVERVIEW	49
What "Object-Oriented Programming" Means	49
The Object Model	50
Properties	51
Methods	51
Collections	52
Try It	53
LESSON 6: VARIABLES, DATA TYPES, AND CONSTANTS	55
What Is a Variable?	55
Assigning Values to Variables	56
Why You Need Variables	56
Data Types	57
Understanding the Different Data Types	57
Declaring a Variable for Dates and Times	58
Declaring a Variable with the Proper Data Type	59

Forcing Variable Declaration	59
Understanding a Variable's Scope	61
Local Macro Level Only	62
Module Level	62
Application Level	63
Constants	63
Try It	64
Lesson Requirements	64
Step-by-Step	64
LESSON 7: UNDERSTANDING OBJECTS AND COLLECTIONS	67
Workbooks	67
Cells and Ranges	69
SpecialCells	70
Try It	71
Lesson Requirements	71
Step-by-Step	71
LESSON 8: WORKING WITH RANGES	75
Working with Contiguously Populated Ranges	75
Using the Cells Property	76
Using CurrentRegion	76
Working with Noncontiguously Populated Ranges	77
Using Range with Several Cells	77
Using OFFSET	78
Using RESIZE	78
Identifying a Data Range	79
Identifying the UsedRange	79
Finding the Dynamic Last Rows and Columns	80
Identifying Where the Range Starts and Ends When No Start or End Point Is Known	81
Try It	82
Lesson Requirements	82
Hints	82
Step-by-Step	82
LESSON 9: MAKING DECISIONS WITH VBA	85
Understanding Logical Operators	85
AND	86
OR	86
NOT	87

Choosing Between This or That	88
If...Then	88
If...Then...Else	89
If...Then...Elseif	90
IIF	90
Select Case	91
Getting Users to Make Decisions	92
Message Boxes	93
Input Boxes	94
Try It	94
Lesson Requirements	95
Hints	95
Step-by-Step	95

PART III: BEYOND THE MACRO RECORDER: WRITING YOUR OWN CODE

LESSON 10: REPEATING ACTIONS WITH LOOPS	101
What Is a Loop?	101
Types of Loops	102
Do...While	106
Do...Until	107
Do...Loop While	109
Do...Loop Until	109
While...Wend	110
Nesting Loops	110
Try It	111
Lesson Requirements	111
Hints	111
Step-by-Step	111
LESSON 11: PROGRAMMING FORMULAS	113
Understanding A1 and R1C1 References	113
Getting Started with a Few One-Liners	114
Comparing the Interface of A1 and R1C1 Styles	115
Toggling between A1 and R1C1 Style Views	116
Programming Your Formula Solutions with VBA	118
Using a Mixed Reference to Fill Empty Cells with the Value from Above	118
Using a Named Range with Relative, Mixed, and Absolute References	119

Programming an Array Formula	120
Summing Lists of Different Sizes along a Single Row	122
Try It	124
Lesson Requirements	124
Step-by-Step	125
LESSON 12: WORKING WITH ARRAYS	127
What Is an Array?	127
What Arrays Can Do for You	128
Declaring Arrays	129
The Option Base Statement	130
Boundaries in Arrays	132
Declaring Arrays with Fixed Elements	132
Declaring Dynamic Arrays with ReDim and Preserve	133
Try It	134
Lesson Requirements	134
Step-by-Step	135
LESSON 13: AUTOMATING PROCEDURES WITH WORKSHEET EVENTS	137
What Is an Event?	137
Worksheet Events: An Overview	138
Where Does the Worksheet Event Code Go?	138
Enabling and Disabling Events	140
Examples of Common Worksheet Events	141
Worksheet_Change Event	141
Worksheet_SelectionChange Event	141
Worksheet_BeforeDoubleClick Event	142
Worksheet_Before RightClick Event	142
Worksheet_FollowHyperlink Event	142
Worksheet_Activate Event	143
Worksheet_Deactivate Event	144
Worksheet_Calculate Event	144
Worksheet_PivotTableUpdate Event	144
Try It	144
Lesson Requirements	145
Step-by-Step	145
LESSON 14: AUTOMATING PROCEDURES WITH WORKBOOK EVENTS	149
Workbook Events: An Overview	149
Where Does the Workbook Event Code Go?	149

Entering Workbook Event Code	151
Examples of Common Workbook Events	153
Workbook_Open Event	153
Workbook_BeforeClose Event	154
Workbook_Activate Event	154
Workbook_Deactivate Event	154
Workbook_SheetChange Event	154
Workbook_SheetSelectionChange Event	155
Workbook_SheetBeforeDoubleClick Event	155
Workbook_SheetBeforeRightClick Event	156
Workbook_SheetPivotTableUpdate Event	156
Workbook_NewSheet Event	156
Workbook_BeforePrint Event	157
Workbook_SheetActivate Event	157
Workbook_SheetDeactivate Event	157
Workbook_BeforeSave Event	158
Try It	158
Lesson Requirements	158
Step-by-Step	158
LESSON 15: HANDLING DUPLICATE ITEMS AND RECORDS	161
Deleting Rows Containing Duplicate Entries	161
Deleting Rows with Duplicates in a Single Column	161
Deleting Rows with Duplicates in More Than One Column	164
Deleting Some Duplicates and Keeping Others	165
Working with Duplicate Data	167
Compiling a Unique List from Multiple Columns	167
Updating a Comment to List Unique Items	169
Selecting a Range of Duplicate Items	171
Inserting an Empty Row at Each Change in Items	172
Try It	173
Lesson Requirements	174
Hints	174
Step-by-Step	174
LESSON 16: USING EMBEDDED CONTROLS	181
Working with Form Controls and ActiveX Controls	181
The Forms Toolbar	182
Buttons	183
Using Application.Caller with Form Controls	184

The Control Toolbox	186
CommandButtons	187
Try It	191
Lesson Requirements	192
Step-by-Step	192
LESSON 17: PROGRAMMING CHARTS	199
Adding a Chart to a Chart Sheet	200
Adding an Embedded Chart to a Worksheet	202
Moving a Chart	204
Looping Through All Embedded Charts	206
Deleting Charts	207
Renaming a Chart	208
Try It	208
Lesson Requirements	208
Step-by-Step	209
LESSON 18: PROGRAMMING PIVOTTABLES AND PIVOTCHARTS	213
Creating a PivotTable Report	213
Hiding the PivotTable Field List	217
Formatting Numbers in the Values Area	219
Pivoting Your Data	222
Creating a PivotChart	223
Understanding PivotCaches	226
Manipulating PivotFields in VBA	230
Manipulating PivotItems with VBA	231
Creating a PivotTables Collection	231
Try It	232
Lesson Requirements	232
Step-by-Step	233
LESSON 19: USER-DEFINED FUNCTIONS	237
What Is a User-Defined Function?	237
Characteristics of User-Defined Functions	237
Anatomy of a UDF	238
UDF Examples That Solve Common Tasks	239
Summing Numbers in Colored Cells	239
Extracting Numbers or Letters from an Alphanumeric String	241
Extracting the Address from a Hyperlink	242

Volatile Functions	243
Returning the Name of the Active Worksheet and Workbook	243
UDFs with Conditional Formatting	244
Calling Your UDF from a Macro	245
Adding a Description to the Insert Function Dialog Box	246
Try It	248
Lesson Requirements	248
Step-by-Step	249
LESSON 20: DEBUGGING YOUR CODE	251
What Is Debugging?	251
What Causes Errors?	252
Weapons of Mass Debugging	254
The Debug Toolbar	254
Trapping Errors	264
Error Handler	264
Bypassing Errors	265
Try It	266
Lesson Requirements	267
Hints	267
Step-by-Step	267
PART IV: ADVANCED PROGRAMMING TECHNIQUES	
LESSON 21: CREATING USERFORMS	271
What Is a UserForm?	271
Creating a UserForm	272
Designing a UserForm	273
Adding Controls to a UserForm	274
Showing a UserForm	280
Where Does the UserForm's Code Go?	281
Closing a UserForm	281
Unloading a UserForm	282
Hiding a UserForm	283
Try It	283
Lesson Requirements	283
Step-by-Step	283
LESSON 22: USERFORM CONTROLS AND THEIR FUNCTIONS	285
Understanding the Frequently Used UserForm Controls	285
CommandButtons	286
Labels	287

TextBoxes	288
ListBoxes	290
ComboBoxes	292
Check Boxes	295
Option Buttons	296
Frames	298
MultiPages	300
Try It	301
Lesson Requirements	301
Step-by-Step	301
LESSON 23: ADVANCED USERFORMS	305
The UserForm Toolbar	305
Modal versus Modeless	306
Disabling the UserForm's Close Button	307
Maximizing Your UserForm's Size	308
Selecting and Displaying Photographs on a UserForm	308
Unloading a UserForm Automatically	309
Pre-sorting the ListBox and ComboBox Items	310
Populating ListBoxes and ComboBoxes with Unique Items	312
Displaying a Real-Time Chart in a UserForm	314
Try It	315
Lesson Requirements	315
Step-by-Step	315
LESSON 24: CLASS MODULES	321
What Is a Class?	321
What Is a Class Module?	322
Creating Your Own Objects	323
An Important Benefit of Class Modules	323
Creating Collections	326
Class Modules for Embedded Objects	326
Try It	330
Lesson Requirements	330
Step-by-Step	330
LESSON 25: ADD-INS	335
What Is an Excel Add-In?	335
Creating an Add-In	336
Converting a File to an Add-In	341
Installing an Add-In	342

Creating a User Interface for Your Add-In	346
Changing the Add-In's Code	348
Closing Add-Ins	349
Removing an Add-In from the Add-Ins List	349
Try It	350
Lesson Requirements	350
Step-by-Step	350
LESSON 26: MANAGING EXTERNAL DATA	353
Creating QueryTables from Web Queries	353
Creating a QueryTable for Access	356
Using Text Files to Store External Data	359
Try It	361
Lesson Requirements	362
Step-by-Step	362
LESSON 27: DATA ACCESS WITH ACTIVEX DATA OBJECTS	365
Introducing ADO	365
The Connection Object	367
The Recordset Object	367
The Command Object	368
An Introduction to Structured Query Language (SQL)	368
The SELECT Statement	369
The INSERT Statement	369
The UPDATE Statement	370
The DELETE Statement	370
Try It	371
LESSON 28: IMPRESSING YOUR BOSS (OR AT LEAST YOUR FRIENDS)	373
Selecting Cells and Ranges	373
Coloring the Active Cell, Row, or Column	373
Coloring the Current and Prior Selected Cells	375
Filtering Dates	376
Filtering between Dates	376
Filtering for Dates before Today's Date	378
Filtering for Dates after Today's Date	378
Deleting Rows for Filtered Dates More Than Three Years Ago	378
Setting Page Breaks for Specified Areas	379
Using a Comment to Log Changes in a Cell	380

Using the Windows API with VBA	381
Clearing the Clipboard	381
Emptying the Recycle Bin	382
Scheduling Your Workbook for Suicide	382
Try It	382
Lesson Requirements	383
Hints	383
Step-by-Step	383
PART V: INTERACTING WITH OTHER OFFICE APPLICATIONS	
LESSON 29: OVERVIEW OF OFFICE AUTOMATION FROM EXCEL	391
Why Automate Another Application?	391
Understanding Office Automation	392
Early Binding	392
Late Binding	394
Which One Is Better?	394
Try It	395
Lesson Requirements	395
Hints	395
Step-by-Step	395
LESSON 30: WORKING WITH WORD FROM EXCEL	399
Activating a Word Document	399
Activating the Word Application	399
Opening and Activating a Word Document	400
Creating a New Word Document	402
Copying an Excel Range to a Word Document	402
Printing a Word Document from Excel	403
Importing a Word Document to Excel	404
Try It	405
Lesson Requirements	406
Step-by-Step	406
LESSON 31: WORKING WITH OUTLOOK FROM EXCEL	409
Opening Outlook	409
Composing an E-mail in Outlook from Excel	410
Creating a MailItem Object	410
Transferring an Excel Range to the Body of Your E-mail	411

Putting It All Together	413
E-mailing a Single Worksheet	415
Try It	415
Lesson Requirements	415
Step-by-Step	415
LESSON 32: WORKING WITH ACCESS FROM EXCEL	419
Adding a Record to an Access Table	419
Exporting an Access Table to an Excel Spreadsheet	423
Creating a New Table in Access	426
Try It	427
Lesson Requirements	427
Step-by-Step	427
LESSON 33: WORKING WITH POWERPOINT FROM EXCEL	431
Creating a New PowerPoint Presentation	431
Copying a Worksheet Range to a PowerPoint Slide	432
Copying Chart Sheets to PowerPoint Slides	433
Running a PowerPoint Presentation from Excel	435
Try It	436
Lesson Requirements	436
Step-by-Step	436
INDEX	441

INTRODUCTION

CONGRATULATIONS ON MAKING TWO EXCELLENT CHOICES! You want to learn programming for Microsoft Excel with Visual Basic for Applications (VBA), and you've purchased this book to teach you. Excel is the most powerful and widely used spreadsheet application in the world. VBA enables you to become much more productive and efficient, while getting your everyday Excel tasks done more quickly and with fewer errors. You'll gain a programming skill that is in high demand, which will improve your value in the workplace and your marketability when searching for employment.

This book covers VBA from the ground up, and assumes you have never programmed Excel before. If you've never recorded or written an Excel macro, this book shows you how. If you've worked with VBA before, this book has examples of programming techniques you might not have seen. The instruction and examples in this book teach VBA concepts that range in levels from fundamental to advanced. The techniques in this book apply just as well to the Excel business power user as to the keeper of the family budget.

VBA is the programming language for Microsoft's popular Office suite of applications, including Excel, Word, Access, PowerPoint, and Outlook. A full section of this book explains how to control each of those applications from Excel with VBA. By the time you complete this book, you will have learned how to record, write, and run your own macros. You'll learn how to make VBA run itself by programming Excel to monitor and respond to users' actions, and how to create friendly, customized interfaces that the users of your workbooks will enjoy.

The future of VBA is solid. Microsoft has confirmed time and again that VBA will be supported in versions of Excel into the foreseeable future, and the programming skills you learn in this book will serve you throughout your career. You'll be able to apply the principles you learn in this book to other tasks that can be automated in Excel and Microsoft's other Office applications. VBA is an enormous programming language, and when combined with Excel, using it is an ongoing, rewarding process of learning something new every day. With this book as your entry into the world of VBA programming, you are well on your way.

WHO THIS BOOK IS FOR

This book is for Excel users who have never programmed Excel before. You are an Excel user who has been doing a frequent task manually, and you are ready to automate the task with VBA. You might also be a job seeker, and you want to improve your chances of being hired in this difficult job market by learning a valuable skill. Whether your Excel tasks are large or small, this book is for you. You find out how to use VBA to automate your work by doing anything from recording a simple one-line macro to writing a complex program with a customized, user-friendly interface that will look nothing like Excel. This book has something for everyone, but especially for the person who wants to dive right into VBA from square one and learn to use its powerful programming tools.

WHAT THIS BOOK COVERS

This book contains 33 lessons, which are broken into five parts:

- **Part I, Understanding the BASICS:** Part I includes Lessons 1 to 4, introducing you to VBA by providing a historical background and a discussion of what VBA is and what it can do for you. This part familiarizes you with the Macro Recorder and the Visual Basic Editor, where VBA code is maintained.
- **Part II, Diving Deeper Into VBA:** Part II includes Lessons 5 to 9, which discuss VBA topics including an overview of object-oriented programming, variable declaration, objects and collections, arrays, and options for decision-making.
- **Part III, Beyond the Macro Recorder: Writing Your Own Code:** Part III includes Lessons 10 to 20. You learn how to write your own macros without help from the Macro Recorder. You become familiar with loops, event programming at the workbook and worksheet levels, charts, PivotTables, user-defined functions, and embedded controls. You learn to program formulas and how to debug your VBA code.
- **Part IV, Advanced Programming Techniques:** Part IV includes Lessons 21 to 28, and deals with the more advanced topics of UserForms, class modules, add-ins, retrieving external data, and various examples of programming Excel to achieve solutions you might not have thought possible.
- **Part V, Interacting with Other Office Applications:** Part V includes Lessons 29 to 33, dealing with how to control Word, Outlook, Access, and PowerPoint from Excel.

HOW THIS BOOK IS STRUCTURED

My main principle in this book is to teach you what you need to know in VBA. I tried to write this book as if you and I were sitting down in front of your computer, and I was explaining Excel and VBA's technical concepts in an informal tutorial session. The book is structured such that each lesson teaches you the theory of a topic, followed by one or more coded examples, with plenty of screenshots and notes to help you follow along. To avoid redundancy of instruction, the lessons build on each other, so the later chapters assume you've read, or are already familiar with, the material discussed in earlier lessons. I strongly recommend that you watch the videos, which you can find at www.wrox.com/go/excelvba24hour. You will get more out of them than you might imagine because they include bonus information about Excel, such as tips and tricks that will help you manage your workbooks with greater ease and efficiency.

WHAT YOU NEED TO USE THIS BOOK

What you need is this book and a fully installed version of Microsoft Office. If you only have Excel installed, that will suffice for lessons up to and including Lesson 28. Lessons 29 to 33 deal with

controlling other Office applications from Excel. VBA ships with Excel, so you already have all the programming tools you need when you installed VBA with Office. The version of your Windows operating system is not important.

In many examples, different versions of Excel are represented, with Excel’s latest version at this writing—version 2013—shown most frequently. If you are using Excel version 2003 or before, you can complete almost all the examples in this book, but it will be easier for you to follow along by using a version starting with 2007—ideally with 2010 or 2013. Almost everything discussed in this book has VBA example code to go along with it, with comments in the code (lines of text in VBA code that start with an apostrophe) that explain what the code is doing, and why. Plenty of screenshots help you see beforehand what to expect, and help you after you’ve tested your code to confirm you followed the steps correctly.

You need one other thing, which only you can control, and that is a quiet period of time for yourself so you can read this book and view its video Try It lessons uninterrupted. Everyone studies and retains new material differently, and we all live in a busy world. But do what you can to carve out some “you time” as you make your way through the book. You’ll find a lot of useful material that will lead you to think of other situations you typically encounter in Excel that can be solved with the concepts you’ll be learning.

CONVENTIONS

To help you get the most from the text and keep track of what’s happening, we’ve used a number of conventions throughout the book.

WARNING *Boxes like this one hold important, not-to-be forgotten information that is directly relevant to the surrounding text.*

NOTE *Notes, tips, hints, tricks, and asides to the current discussion are offset and placed in italic like this.*

As for styles in the text:

- We *highlight* new terms and important words when we introduce them.
- We show filenames, URLs, and code within the text like so: `persistence.properties`.
- We present code like this:

We use a monofont type with no highlighting for most code examples.

We use bold to emphasize code that's particularly important in the present context.

- Text that you need to enter as you work through the Try It sections is written as bold code, as shown here:

Name it **cmdExit** and caption it as **Exit**.

Source Code

As you work through the examples in this book, you may choose either to type in all the code manually or to use the source code files that accompany the book. All of the source code used in this book is available for download at www.wrox.com/go/excelvba24hour. The code snippets from the source code are accompanied by a download icon and note indicating the name of the program so you know it's available for download and can easily locate it in the download file. Once at the site, simply locate the book's title (either by using the Search box or by using one of the title lists) and click the Download Code link on the book's detail page to obtain all the source code for the book.

After you download the code, just unzip the file using WinZip or a similar tool. Alternatively, you can go to the main Wrox code download page at <http://www.wrox.com/dynamic/books/download.aspx> to see the code available for this book and all other Wrox books.

ERRATA

We make every effort to ensure that there are no errors in the text or in the code. However, no one is perfect, and mistakes do occur. If you find an error in one of our books, like a spelling mistake or faulty piece of code, we would be very grateful for your feedback. By sending in errata you may save another reader hours of frustration and at the same time you will be helping us provide even higher quality information.

To find the errata page for this book, go to <http://www.wrox.com> and locate the title using the Search box or one of the title lists. Then, on the book details page, click the Book Errata link. On this page you can view all errata that has been submitted for this book and posted by Wrox editors. A complete book list including links to each book's errata is also available at www.wrox.com/misic-pages/booklist.shtml.

If you don't spot "your" error on the Book Errata page, go to www.wrox.com/contact/techsupport.shtml and complete the form there to send us the error you have found. We'll check the information and, if appropriate, post a message to the book's errata page and fix the problem in subsequent editions of the book.

P2P.WROX.COM

For author and peer discussion, join the P2P forums at p2p.wrox.com. The forums are a web-based system for you to post messages relating to Wrox books and related technologies and interact with other readers and technology users. The forums offer a subscription feature to e-mail you topics of interest of your choosing when new posts are made to the forums. Wrox authors, editors, other industry experts, and your fellow readers are present on these forums.

At <http://p2p.wrox.com> you will find a number of different forums that will help you not only as you read this book, but also as you develop your own applications. To join the forums, just follow these steps:

1. Go to p2p.wrox.com and click the Register link.
2. Read the terms of use and click Agree.
3. Complete the required information to join as well as any optional information you wish to provide and click Submit.
4. You will receive an e-mail with information describing how to verify your account and complete the joining process.

NOTE *You can read messages in the forums without joining P2P but in order to post your own messages, you must join.*

Once you join, you can post new messages and respond to messages other users post. You can read messages at any time on the web. If you would like to have new messages from a particular forum e-mailed to you, click the Subscribe to this Forum icon by the forum name in the forum listing.

For more information about how to use the Wrox P2P, be sure to read the P2P FAQs for answers to questions about how the forum software works as well as many common questions specific to P2P and Wrox books. To read the FAQs, click the FAQ link on any P2P page.

Excel® VBA 24-Hour Trainer

PART I

Understanding the BASICs

- ▶ **LESSON 1:** Introducing VBA
- ▶ **LESSON 2:** Getting Started with Macros
- ▶ **LESSON 3:** Introducing the Visual Basic Editor
- ▶ **LESSON 4:** Working in the VBE

1

Introducing VBA

Welcome to your first lesson in *Excel VBA 24-Hour Trainer!* A good place to start is at the beginning, where you'll find it useful to get an understanding of where Visual Basic for Applications (VBA) comes from and what VBA is today. After you get a feel for how VBA fits into the overall Excel universe, you find out how to use VBA to manipulate Excel in ways you might never have thought possible.

WHAT IS VBA?

VBA is a programming language created by Microsoft to automate operations in applications that support it, such as Excel. VBA is an enormously powerful tool that enables you to control Excel in countless ways that you cannot do—or would not want to do—manually.

In fact, VBA is also the language that manipulates Microsoft Office applications in Access, Word, PowerPoint, and Outlook. For the purposes here, VBA is the tool you use to develop macros and manipulate objects to control Excel and to control other Office applications from Excel.

You do not need to purchase anything more than the Office suite (or the individual application) to also own VBA. If you have Excel on your computer, you have VBA on your computer.

WHAT IS A “MACRO,” ANYWAY?

Back in the day, a programming language was often called a “macro language” if its capabilities included the automation of a sequence of commands in spreadsheet or word-processing applications. With Microsoft’s release of Office 5, VBA set a new bar for how robust a programming language can be, with capabilities extending far beyond those of earlier programming languages, such as the ability to create and control objects within Excel or to have access to disk drives and networks.

continues

continued

So VBA is a programming language, and it is also a macro language. Confusion of terminology arises when referring to VBA code that is a series of commands written and executed in Excel. Is it a macro, a procedure, or a program? Microsoft commonly refers to its VBA procedures as macros, so that's good enough for me to call them macros also. Outside of a few exceptions that I explain when the time comes, I refer to VBA procedures as macros.

A BRIEF HISTORY OF VBA

VBA is a present-day dialect of the BASIC (Beginner's All-purpose Symbolic Instruction Code) programming language that was developed in the 1960s. BASIC became widely used in many software applications throughout the next two decades because it was easy to learn and understand.

Over the years, BASIC has evolved and improved in response to advancing technology and increased demands by its users for greater programming flexibility. In 1985, Microsoft released a much richer version of BASIC, named QuickBASIC, which boasted the most up-to-date features found in programming languages of the day. In 1992, Microsoft released Visual Basic for Windows, designed to work within the burgeoning Windows environment.

Meanwhile, various software publishers were making their own enhancements to BASIC for their products' programming languages, resulting in a wide and confusing range of functionality and commands among software applications that were using BASIC. Microsoft recognized the need for developing a standardized programming language for its software products, and created Visual Basic for Applications.

VBA was first released by Microsoft with Excel 5 in the Office 1995 suite. Since then, VBA has become the programming language for Microsoft's other popular Office applications, as well as for external software customers of Microsoft to whom VBA has been licensed for use.

THERE'S A BIG DIFFERENCE BETWEEN VB AND VBA!

With all the acronyms bandied about in the world of computing, it's easy to get some terms confused. VB stands for Visual Basic, and it is not the same as VBA. Though both VB and VBA are programming languages derived from BASIC and created by Microsoft, they are otherwise very different.

VB is a language that enables you to create standalone executable applications that do not even require its users to have Office or Excel loaded onto their computers. VBA cannot create standalone applications, and it exists within a host application such as Excel and the workbook containing the VBA code. For a VBA macro to run, its host application workbook must be open. This book is about VBA and how it controls Excel.

WHAT VBA CAN DO FOR YOU

Everyone reading this book uses Excel for their own needs, such as financial budgeting, forecasting, analyzing scientific data, creating invoices, or charting the progress of their favorite football team. One thing all readers have in common is the need to automate some kind of frequently encountered task that is either too time-consuming or too cumbersome to continue doing manually. That's where VBA comes in.

The good news is that utilizing VBA does not mandate that you first become a world-class professional programmer. Many VBA commands are at your disposal, and are relatively easy to implement and customize for your everyday purposes.

Anything you can do manually you can do with VBA, but VBA enables you to do it faster and with a minimized risk of human error. Many things that Excel does not allow you to do manually, you can do with VBA. The following sections describe a handful of examples of what VBA can do for you.

Automating a Recurring Task

If you find yourself needing to produce weekly or monthly sales and expense reports, a macro can create them in no time flat, in a style and format you (and more importantly, your boss) will be thrilled with. And if the source data changes later that day and you need to produce the updated report again, no problem—just run the macro again!

Automating a Repetitive Task

When faced with needing to perform the same task on every worksheet in your workbook, or in every workbook in a particular file folder, you can create a macro to “loop” through each object and do the deed. You find out how to repeat actions with various looping methods in Lesson 10. Figure 1-1 shows an example of worksheets that were sorted in alphabetical order by a macro that looped through each tab name, repositioning each sheet in the process.

Running a Macro Automatically if Another Action Takes Place

In some situations, you want a macro to run automatically so you don’t have to worry about remembering to run it yourself. For example, to automatically refresh a pivot table the moment its source data changes, you can monitor those changes with VBA, ensuring that your pivot table always displays real-time results. This is called “event” programming, which is cool stuff, and is discussed in Lessons 13 and 14.

An event can also be triggered and programmed anytime a cell or range of cells is selected. A common request I’ve received from Excel users is to highlight the active cell, or the row and column belonging to the active cell, automatically when a cell is selected. Figure 1-2 shows three options to easily locate your active cell as you traverse your worksheet.

Worksheets before sorted by tab name.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Sales report for Jack Jones (numbers in thousands)													
2														
3		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Total
4	Widgets	34	67	76	13	62	67	87	76	40	8	54	34	618
5	Whistles	90	29	83	80	50	37	18	34	51	34	84	45	635
6	Wombats	43	71	83	12	68	73	73	50	74	15	37	43	642
7	Total	167	167	242	105	180	177	178	160	165	57	175	122	1895
8														
	Jones	Garcia	Zimmer	Smith	Brown	Lee	Adams	Miller						

Worksheets after sorted by tab name.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Sales report for Amy Adams (numbers in thousands)													
2														
3		Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec	Total
4	Widgets	38	58	11	76	47	83	2	44	28	84	24	45	540
5	Whistles	55	66	13	70	45	6	63	56	88	90	8	63	623
6	Wombats	80	49	10	19	98	70	1	54	52	75	40	36	584
7	Total	173	173	34	165	190	159	66	154	168	249	72	144	1747
8														
	Adams	Brown	Carter	Garcia	Jones	Lee	Miller	Smith	Zimmer					

FIGURE 1-1

Highlight the active cell.						Highlight active row and column.						Highlight the row and column in the active cell's current region.						
A	B	C	D	E	F	A	B	C	D	E	F	A	B	C	D	E	F	
1	Atlas Programming					1	Atlas Programming					1	Atlas Programming					
2	Budget Year 2015					2	Budget Year 2015					2	Budget Year 2015					
3						3						3						
4		Q1	Q2	Q3	Q4	4		Q1	Q2	Q3	Q4	4		Q1	Q2	Q3	Q4	
5	Income	893	651	757	462	5	Income	893	651	757	462	5	Income	893	651	757	462	
6	Expenses	849	466	556	444	6	Expenses	849	466	556	444	6	Expenses	849	466	556	444	
7	Net Gain	44	185	201	18	7	Net Gain	44	185	201	18	7	Net Gain	44	185	201	18	
8	Capital	24	100	188	16	8	Capital	24	100	188	16	8	Capital	24	100	188	16	
9	Net Profit	20	85	13	2	9	Net Profit	20	85	13	2	9	Net Profit	20	85	13	2	
10						10						10						
11						11						11						
12	Month	Name	Sales	12	Month	Name	Sales	12	Month	Name	Sales	12	Month	Name	Sales	12	Month	
13	January	Bill	903	13	January	Bill	903	13	January	Bill	903	13	January	Bill	903	13	January	
14	February	Bob	959	14	February	Bob	959	14	February	Bob	959	14	February	Bob	959	14	February	
15	March	Tom	707	15	March	Tom	707	15	March	Tom	707	15	March	Tom	707	15	March	
16	April	Mike	618	16	April	Mike	618	16	April	Mike	618	16	April	Mike	618	16	April	
17	May	Jim	796	17	May	Jim	796	17	May	Jim	796	17	May	Jim	796	17	May	
18	June	Sophia	550	18	June	Sophia	550	18	June	Sophia	550	18	June	Sophia	550	18	June	
19	July	Angie	49	19	July	Angie	49	19	July	Angie	49	19	July	Angie	49	19	July	
20	August	Bella	523	20	August	Bella	523	20	August	Bella	523	20	August	Bella	523	20	August	
21	September	Frank	654	21	September	Frank	654	21	September	Frank	654	21	September	Frank	654	21	September	
22	October	Joe	919	22	October	Joe	919	22	October	Joe	919	22	October	Joe	919	22	October	
23	November	Pete	532	23	November	Pete	532	23	November	Pete	532	23	November	Pete	532	23	November	
24	December	Alex	727	24	December	Alex	727	24	December	Alex	727	24	December	Alex	727	24	December	
25				25				25				25				25		

FIGURE 1-2

Creating Your Own Worksheet Functions

You can create your own worksheet functions, known as *user-defined functions*, to handle custom calculations that Excel's built-in functions do not provide, or would be too complicated to use even if such native functions were available. For example, later in the book you see how to add up numbers in cells that are formatted a certain color. UDFs, as these custom functions are called, are covered in Lesson 19, "User-Defined Functions."

Simplifying the Workbook's Look and Feel for Other Users

When you create a workbook for others to use, there will inevitably be users who know little to nothing about Excel, but who will still need to work in that file. You can build a customized interface with user-friendly menus and informational pop-up boxes to guide your novice users throughout their activities in the workbook. You might be surprised at how un-Excel-looking an Excel workbook can be, with VBA providing a visually comfortable and interactive experience for users unfamiliar with Excel, enabling them to get their work done. Figure 1-3 shows an example of accomplishing this with UserForms, which are discussed in Lessons 21, 22, and 23.



FIGURE 1-3

Controlling Other Office Applications from Excel

If you create narrative reports in Word that require an embedded list of data from Excel, or if you need to import a table from Access into an Excel worksheet, VBA can automate the process. VBA is the programming language for Microsoft's other Office applications, enabling you to write macros

in Excel to perform tasks in those other applications, with the users being none the wiser that they ever left Excel while the macro was running.

As you might imagine, the list of advantages to using VBA could fill the capacity of your average flash drive. The point is, you are sure to have tasks in your everyday dealings with Excel that can be accomplished more quickly and efficiently with VBA, and this book shows you how.

LIABILITIES OF VBA

Although VBA is a tremendously useful and versatile tool, it is not a 100 percent perfect programming language—but then, no programming language anywhere can truthfully claim infallibility. The pros of VBA far outweigh its cons, but learning and using VBA does come with a few objective caveats that you should be aware of:

- With each version release of Excel, Microsoft may add new VBA commands or stop supporting existing VBA commands, sometimes without advance warning. Surprises do happen, as was especially the case when Office 2007 was released with all its added features. Such is life in the world of Excel VBA. You will probably learn of coding errors from people who have upgraded to a newer version and are using the workbook you created in an earlier version.
- VBA does not run uniformly in all computer operating environments. Sometimes, no matter how extensively you test your code and how flawlessly the macros run on your computer as you develop a project, there will be users of your workbook who will eventually report an error in your code. It won't be your fault or VBA's fault, it's just the idiosyncrasies of how programming languages such as VBA mix with various operating systems, Office versions, and network configurations. Debugging your code is the subject of Lesson 20.
- Programming languages, including VBA, are not warmly received by all workplace IT departments. Many companies have set internal policies that forbid employees from downloading malicious software onto workplace computers. This is an understandable concern, but the corporate safety nets are sometimes cast far and wide to include Excel workbooks with VBA code. The tug of war in companies between the security interests of IT and the work efficiency needs of management can determine whether the VBA code you install will actually be allowed for use in some company venues.
- Finally, VBA is a large program. It has thousands of keywords and the language library is only getting larger. Actually, I see this as a good thing, because the more VBA you learn, the more productivity and control you will have with Excel. Just as with any language, be it spoken or programming, there is a level of rolling-up-your-sleeves commitment that'll be needed to learn VBA. Even the longest journey starts with a first step, and this book gets you on your way.

NOTE *VBA has a bright, stable future. An occasional rumor makes the rounds on the Internet, claiming the imminent demise of VBA. Do not believe it. VBA is here to stay, and Microsoft has publicly said so, time and again. The facts are, in 2007, Microsoft closed its VBA licensing program to new customers, and VBA was not supported in the 2008 version of Office for the Mac, though VBA has been supported by Mac versions after that. Microsoft has consistently made very clear its plan for supporting VBA in future versions of Excel for Windows.*

TRY IT

With the introductory nature of this first lesson, there's nothing specific to try with VBA. What you can do is to get a jump on the rest of the lessons in this book by making a list of some of your most frequent everyday manual Excel tasks, especially the dreaded, time-consuming ones you wish would go away. Tasks such as those will become good candidates for you to apply the VBA macros and automated solutions skills that the following lessons will teach you.

REFERENCE *There is no video to accompany this lesson.*

2

Getting Started with Macros

In Lesson 1, you read that VBA is the programming language of Microsoft Excel and that a macro is a sequence of VBA commands to run a task automatically instead of manually. In this lesson, you find out how to create a simple macro, what its code looks like, and a few options for how you can run the macro.

COMPOSING YOUR FIRST MACRO

This lesson leads you through the process of composing a macro to sort and format a range of data. But even before the first line of programming code is written, you need to set up shop by giving yourself easy access to the VBA-related tools you'll be using. The following housekeeping items usually need to be done only once, and it's worth taking the time to do them now if you haven't already done so.

Accessing the VBA Environment

At the time of this writing, Excel is at a unique stage in its ongoing evolution because four of its versions are being used with significant popularity in the Microsoft Office suite of applications. Version 2003 (also known as version 11) was the final Excel version with the traditional menu bar interface of File, Edit, View, and so on. Then came version 2007 (also known as version 12), blazing the trail for Office's new Ribbon interface. Three years later, version 2010 (also known as version 14) was the next release from Redmond. Most recently, version 2013 (also known as version 15) has taken its place among the community of Excel versions that are being used around the world.

As with other tasks you typically do in Excel, the actions you take to create, view, edit, or run VBA code usually start by clicking the on-screen icon relating to that task. Exactly what those VBA-related icons look like, and what you need to do to make them easily accessible to you, depends on the particular version of Excel you are working with.

WHY IS THERE NO VERSION 13?

You probably noticed that the version numbers went from 12 in 2007 to 14 in 2010, making the number 13 conspicuously absent as a version number. This was not an accident; Microsoft purposely skipped the number 13. You'll often notice in elevators of high-rise office buildings and hotels that the floor buttons go from 12 to 14, without a floor number 13. Microsoft recognizes that its Office applications are used globally, and in some cultures, 13 is thought to be an unlucky number. It made good business sense to avoid issues of possible reluctance from consumers upgrading to “Office 13,” or blame for inevitable version bugs by people who believe that 13 is an unlucky number.

To save yourself time and extra mouse clicks, start by making sure that the VBA-related icons you'll be using most frequently are already displayed whenever you open Excel. The following steps are shown for each of today's four most popular versions.

Version 2003 continues to be used by a measurable percentage of individuals and employers worldwide. For versions of Excel up to and including 2003, from your worksheet menu, click View \Rightarrow Toolbars \Rightarrow Visual Basic, as shown in Figure 2-1. This displays the Visual Basic toolbar, as shown in Figure 2-2, which you can dock just as you do with your other toolbars.

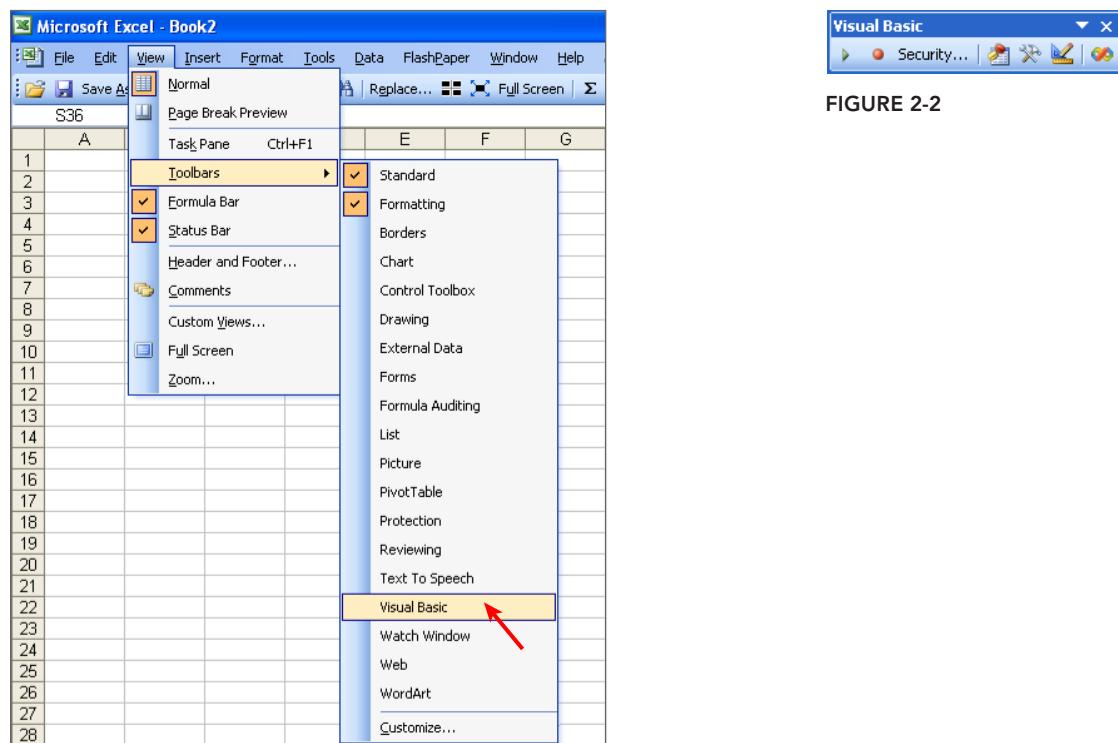


FIGURE 2-2

FIGURE 2-1

For versions of Excel after 2003 (that is, starting with Excel 2007), the Ribbon user interface has replaced the menu interface, resulting in a different look to the VBA-related icons and a different set of steps required to see them.

In versions 2007, 2010, and 2013, these VBA icons are located on the Developer tab. By default, the Developer tab is not automatically displayed along with the other Ribbon tabs. You need to complete a set of one-time steps to show the Developer tab and to keep it visible whenever you open Excel. Although the steps to do this are easy, they are different for each version.

In Excel 2007, do the following:

1. Click the round Office button near the top-left corner of your screen.
2. Click the Excel Options button located at the bottom of that menu, as shown in Figure 2-3.

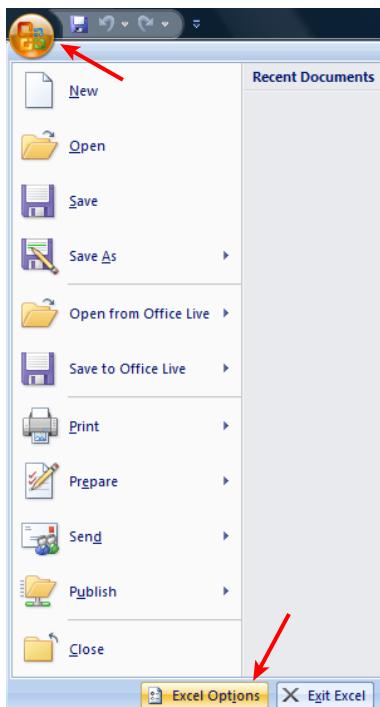


FIGURE 2-3

3. In the Excel Options dialog box, click the Popular item at the upper left, and select the Show Developer tab in the Ribbon option, as shown in Figure 2-4.

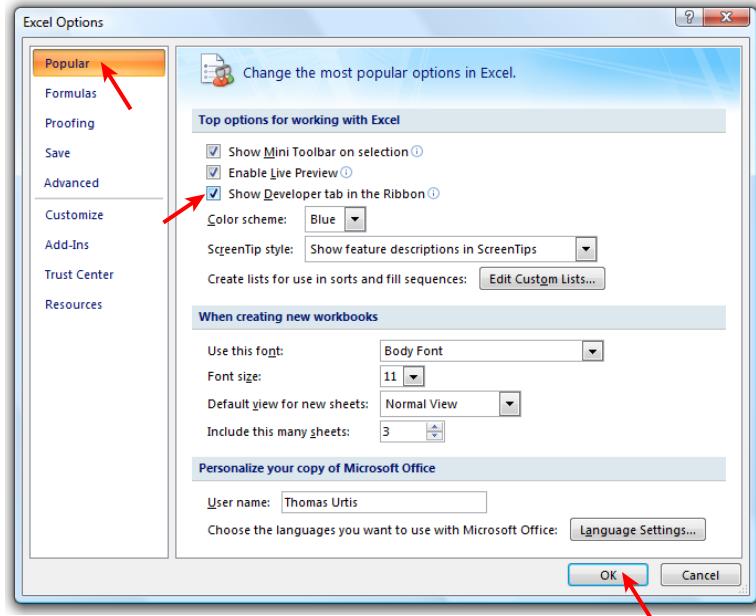


FIGURE 2-4

In Excel versions 2010 and 2013, showing the Developer tab is a bit different. A new Ribbon tab named File has supplanted the Office button. Use the following steps to make the Developer tab visible:

1. Click the File tab and then click the Options button, as shown in Figure 2-5. The Options dialog box opens.

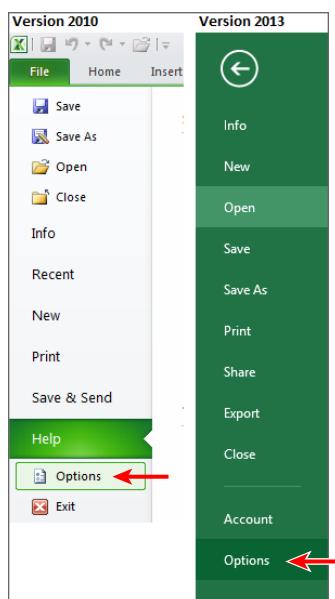


FIGURE 2-5

2. Click the Customize Ribbon item at the left, which displays two vertical lists, as shown in Figure 2-6. Notice that the list on the right has a drop-down menu above it called Customize the Ribbon. Notice that the list on the right has a drop-down menu above it called Customize the Ribbon.

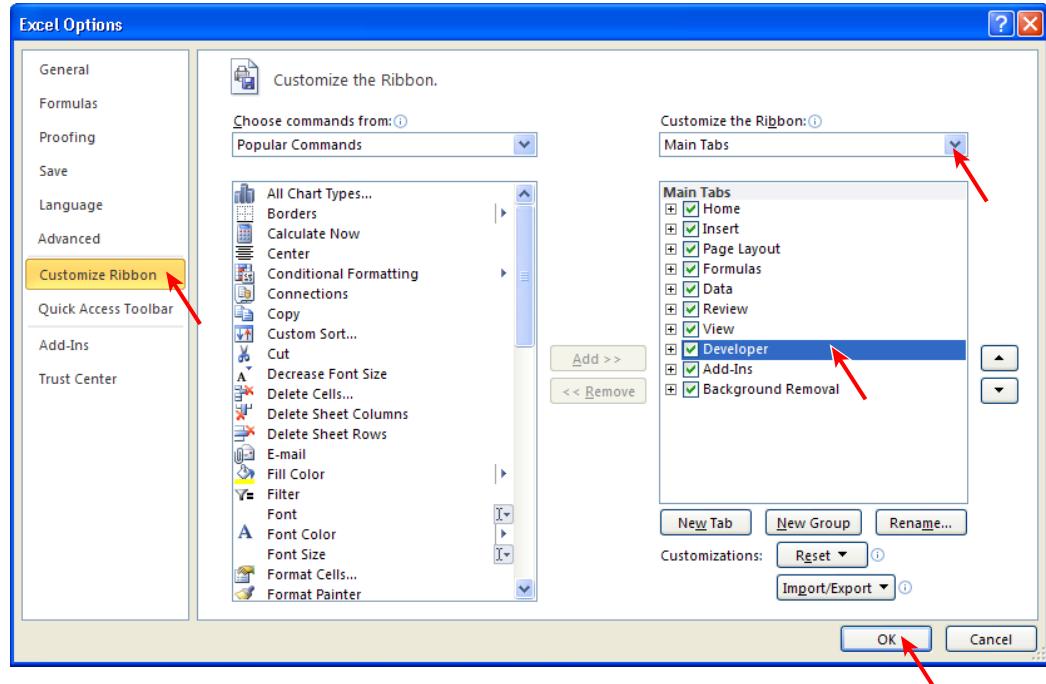


FIGURE 2-6

3. Select the Main Tabs item from the Customize the Ribbon drop-down.
 4. In the list of Main Tabs, select Developer and click OK. You will see the Developer tab in your Ribbon, as shown in Figure 2-7.

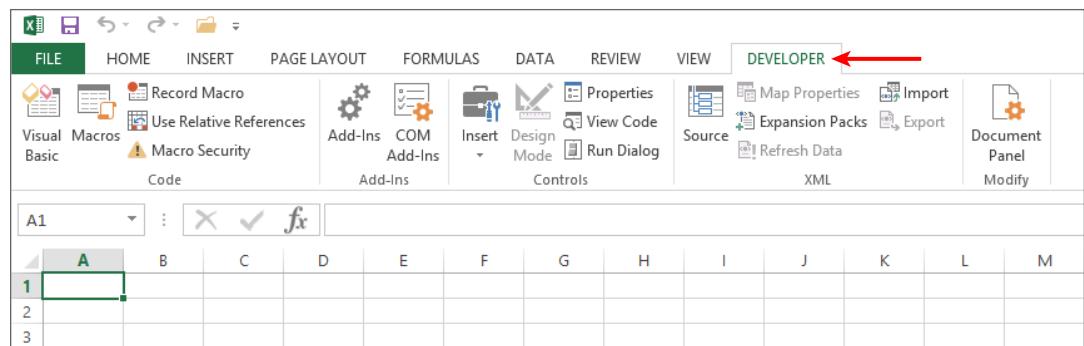


FIGURE 2-7

Using the Macro Recorder

The easiest way to create a macro is to record your worksheet actions using a valuable tool called the Macro Recorder. All you need to do is turn on the Macro Recorder, perform the actions that comprise the task you want to automate, and then turn off the Macro Recorder when you have finished your task. While the Macro Recorder is turned on, every action you do—selecting a cell, entering a number, formatting a range, pretty much everything—is recorded and represented as VBA code in a new macro. As you see later, when you run the macro created by the Macro Recorder, your task is completed automatically, just as if you had done it manually.

The Macro Recorder comes in handy for repetitive (and sometimes mundane) common tasks that you'd rather not have to keep manually doing over and over. For example, say you manage a table of data every day, such as the one shown in Figure 2-8, that shows how many items your company sold in its East, West, North, and South regions.

	A	B	C	D	E	F	G
1	Item	Region	Count				
2	Widgets	North	40931				
3	Wombats	West	48521				
4	Wombats	East	37883				
5	Wallabees	South	82943				
6	Widgets	West	8010				
7	Warlocks	South	65065				
8	Wallabees	West	76781				
9	Wallabees	East	29023				
10	Widgets	North	72685				
11	Widgets	South	2371				
12	Warlocks	East	40681				
13	Wallabees	West	91831				
14							
15							

FIGURE 2-8

The everyday task at hand is to sort the table primarily by Region, then by Item, then by Count. Your boss wants the Item and Region columns to switch places, so that Region occupies column A and Item occupies column B. To improve readability, the numbers in the Count column must be formatted with the thousands comma separator, and the headers for Region, Item, and Count must be bolded. Figure 2-9 shows the finished table, the way your boss wants it.

This is normally a six-step process, which is quite boring, but it's part of your job responsibilities.

	A	B	C	D	E	F	G
1	Region	Item	Count				
2	East	Wallabies	29,023				
3	East	Warlocks	40,681				
4	East	Wombats	37,883				
5	North	Widgets	40,931				
6	North	Widgets	72,685				
7	South	Wallabies	82,943				
8	South	Warlocks	65,065				
9	South	Widgets	2,371				
10	West	Wallabies	76,781				
11	West	Wallabies	91,831				
12	West	Widgets	8,010				
13	West	Wombats	48,521				
14							
15							

FIGURE 2-9

To complete the task you might do this:

1. Insert a new column at column A.
2. Select the Region column, cut it, and paste it to empty column A, to the left of the Item column.
3. Delete the now-empty column from where the Region column was cut.
4. Select range A1:C13 and sort in ascending order by Region, Item, and Count.
5. Select range C2:C13 and format the numbers with the thousands comma separator.
6. Select range A1:C1 and format those cells as Bold.

Not only are these steps monotonous, but also a risk for making honest mistakes due to eventual human error. The good news is that if you perform the necessary steps perfectly for the Macro Recorder, the task can be reduced to a simple mouse click or keyboard shortcut, with VBA doing the grunt work for you.

NOTE *Anytime you create a macro, it's wise to plan ahead about why you are creating the macro, and what you want the macro to do. This is especially important with complex macros, because you want your macros to operate efficiently and accurately, with just the code that's necessary to get the job done properly. By avoiding excessive code, your macros will run faster and be easier to edit or troubleshoot. For example, get your workbook ready beforehand to avoid unnecessary coded actions. Have the worksheet that you'll be working on active, with the range of interest already visible. Mistakes are recorded too! Practice the steps first, so your macro's recorded code is not lengthier than it needs to be.*

Because you know what manual steps are required for this daily task, you are ready to create your macro. The first thing to do is turn on the Macro Recorder. In Excel versions 2003 or before, click the Record Macro button on the Visual Basic toolbar, as shown in Figure 2-10. For later Excel versions, click the Record Macro button in the Code section of the Developer tab on the Ribbon, as shown in Figure 2-11.

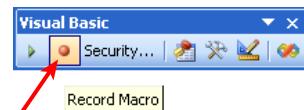


FIGURE 2-10

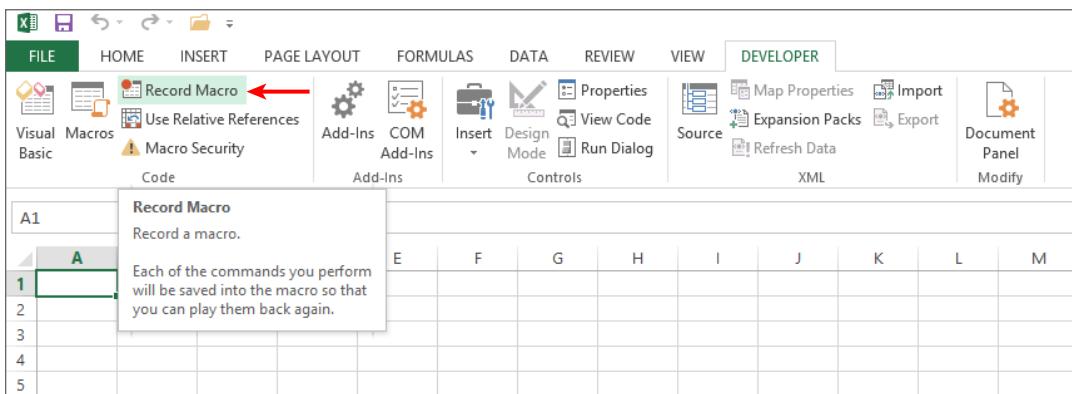


FIGURE 2-11

What you see next looks much like Figure 2-12. A small Record Macro dialog box displays, with default information that only needs your approval by clicking OK to start recording your macro. Resist the temptation to accept the defaults, because now's the time to get into a few good habits.

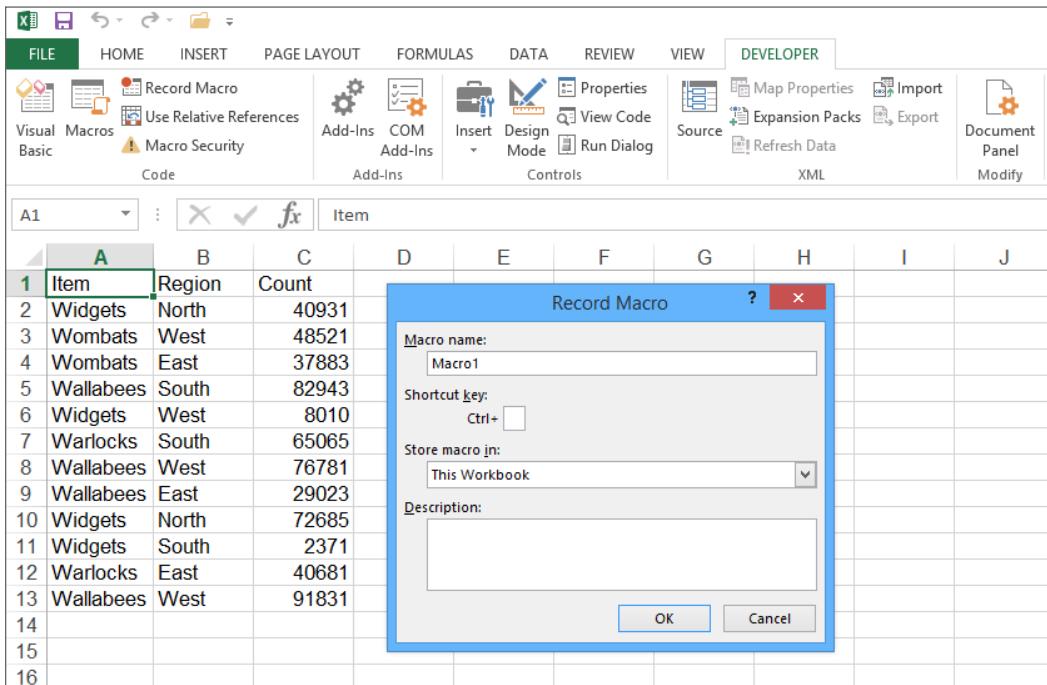


FIGURE 2-12

The Macro Recorder is an excellent teaching tool, and hardly a day goes by when I do not use it in some way. VBA is just too voluminous a programming language to memorize its every keyword and nuance. Often as not, I'll record a macro just to look at the code it produces to learn the proper syntax of a task dealing with some larger macro I am working on. You will find yourself using the Macro Recorder in the same way; it's a terrific source for learning VBA code, as Excel developers of any skill level will attest.

For this example, the macro you are creating is one you will want to keep and use often. A little customization is strongly recommended to help you down the road, when you'll want to remember what the macro does, why you created it, and what optional keyboard shortcut you assigned to run it.

In the Record Macro dialog box, give the macro a meaningful name. Macro names cannot contain spaces, and they cannot begin with a numeral. Because you are the person doing the sorting, and you don't want to make the macro name too long, naming it `mySort` gives the macro more meaning than the default name of `Macro1`.

In Figure 2-12, notice the small box to the right of `Ctrl+` in the Shortcut Key section. You can place any letter of the alphabet in that field, which, when pressed with the `Ctrl` key, will be one method (and a convenient one at that) by which you can run the macro.

NOTE *A shortcut key is not mandatory; in fact, most of your macros will not have one or need one. But if you do want to assign a shortcut key, get into the good habit of assigning it with the `Ctrl+Shift` combination rather than with just the `Ctrl` key. Excel has assigned all 26 letters of the alphabet to serve as built-in shortcuts with the `Ctrl` key for various tasks, and you will do well to avoid overriding that native functionality. For example, `Ctrl+C` is the key combination to copy text. However, if you assign the shortcut key `Ctrl+C` to your macro, you will override the default for that key combination, and will not be able to use `Ctrl+C` to copy text in the workbook containing the macro.*

To take advantage of the Shortcut Key option, click in the Shortcut Key field, press the Shift key, and also press an alphabet key such as the letter S. You will have created the keyboard shortcut `Ctrl+Shift+S`, which will not interfere with any of Excel's significant built-in keyboard shortcuts.

Most macros you record are stored in the workbook you are working with. For now, you can keep the default selection of This Workbook in the Store Macro In field.

Finally, in the Description field, enter a brief but meaningful explanation of what the macro does. When you are finished making these minor changes to the Record Macro dialog box, it looks similar to Figure 2-13. Go ahead and click OK, which turns on the Macro Recorder, and you can proceed to manually perform the steps you want to automate.

FIGURE 2-13

In versions 2003 and earlier, you will see a tiny floating toolbar while the Macro Recorder is on. That is the Stop Recording toolbar, with a Stop Recording button you click when you are finished recording your actions. When you have completed the steps to your task, turn off the Macro Recorder in version 2003 by clicking the Stop Recording button, as shown in Figure 2-14.

If you are working in a later version of Excel, click the Stop Recording button on the Developer tab in the Ribbon, as shown in Figure 2-15. Clicking the Stop Recording button ends the recording session, and you have created your macro.



FIGURE 2-14

FIGURE 2-15

HEY, MY STOP RECORDING BUTTON DISAPPEARED!

If you are using Excel version 2003 or earlier, the Stop Recording toolbar might seem to suddenly disappear from time to time. This is almost always due to unwittingly closing that toolbar by clicking the X close button on its title bar instead of the Stop Recording button. It has happened to the best of us. To show the Stop Recording toolbar again, start to record a new macro, then from the worksheet menu click View → Toolbars → Stop Recording. Click the Stop Recording button to end the macro, and the next time you record a macro, the Stop Recording toolbar will be its normal visible self.

If you are working in version 2007 or later, no worries. The Stop Recording button on the Ribbon does not disappear; it only reverts to Record Macro when clicked.

RUNNING A MACRO

You have many ways to run a macro, most of which are demonstrated in later lessons. As you will see, the method(s) you choose for running your macros may depend on complex reasons such as the workbook design, or may be based on a simpler factor such as what feels most intuitive and convenient for you. To wrap up this lesson, following are a couple of commonly used options for running your macros.

The Macro Dialog Box

When you create recorded macros, their names will appear listed in a dialog box called, appropriately enough, the Macro dialog box. To show the Macro dialog box in version 2003 or earlier, click the Run Macro button on the Visual Basic toolbar, as shown in Figure 2-16.

The title of that button, Run Macro, is something of a misnomer because you are not actually running a macro when you've clicked the button. All you're doing is displaying the Macro dialog box, from which you can run a macro but also edit and examine macros.

In versions later than 2003, the button to click is more logically labeled Macros, as shown in Figure 2-17.

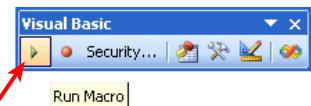


FIGURE 2-16

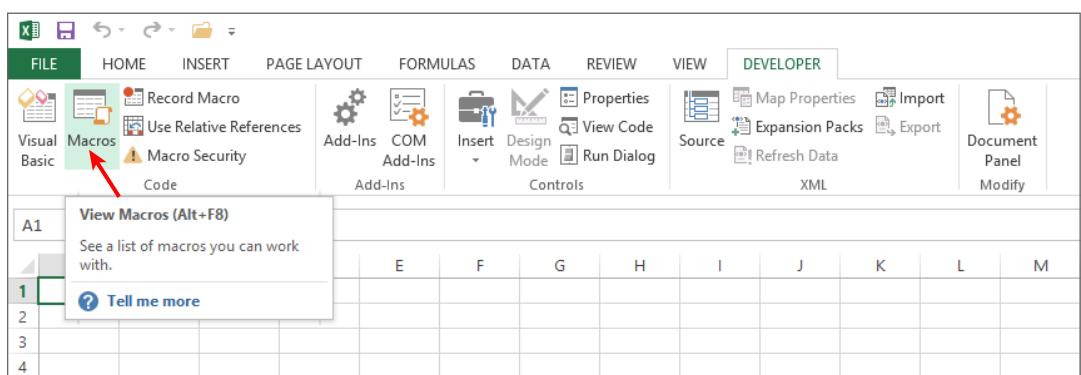


FIGURE 2-17

NOTE Regardless of the Excel version, pressing Alt+F8 displays the Macro dialog box—no mouse clicks needed.

Figure 2-18 shows the Macro dialog box with the one and only mySort macro listed. As you create more macros in this workbook, their names are listed in the Macro dialog box in alphabetical order. To run your macro, select its name in the list and click the Run button, as indicated by the arrows. You could also run the macro by double-clicking its name in the list.

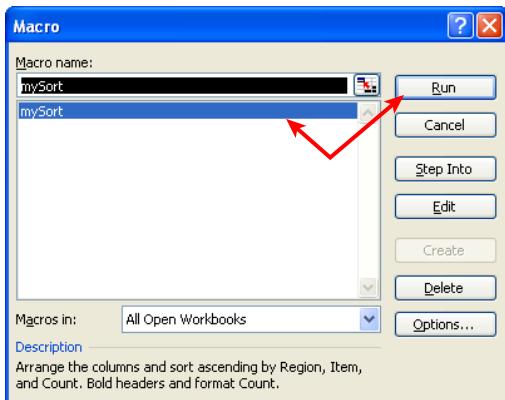


FIGURE 2-18

Shortcut Key

Recall that you assigned the shortcut key Ctrl+Shift+S to this macro at the start of the macro recording process. Because you did that, you do not need to bother with the Macro dialog box if you don't want to; you can run the mySort macro simply by pressing Ctrl+Shift+S.

TRY IT

In this lesson, you practice creating a recorded macro.

Lesson Requirements

To get the sample workbook file, you can download Lesson 2 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

Name your macros with a word or concise phrase that is easy to read and gives an idea about what the macro does. For example, a macro named Print_Expense_Report is more descriptive than Macro5.

Step-by-Step

Start with a worksheet on which some cells contain numbers that were manually entered, and other cells contain numbers produced by formulas, such as in the downloadable budget workbook shown in the video for this lesson. I have a number of steps in this “Try It” lesson to help demonstrate the value of a macro that can automatically perform tedious, recurring manual tasks with a simple keyboard shortcut or click of a button.

Create a macro that fills the manually entered numeric cells with one color, and the formula-containing numeric cells with another color:

1. Click the Record Macro button to turn on the Macro Recorder.
2. In the Record Macro dialog box, name the macro **My_Macro** and assign it the shortcut **Ctrl+Shift+W**.
3. Click OK to start recording your **My_Macro** macro.
4. Click the button above row 1 and to the left of column A to select all the worksheet cells.
5. Show the Format Cells dialog box. Right-click any selected cell and select Format Cells from the menu, or press the **Alt+O+E** keyboard shortcut.
6. In the Format Cells dialog box, click the Fill tab. Click the No Color button and click OK to remove the fill colors from all cells.
7. With all the worksheet cells still selected, press the **F5** key to show the Go To dialog box. Click the Special button.
8. In the Go To Special dialog box, select the option button for Constants, leave the Numbers check box selected, and deselect the check boxes for Text, Logicals, and Errors. Click OK.
9. Repeat Step 5 to show the Format Cells dialog box.
10. In the Format Cells dialog box, click the Fill tab, select a color from the palette, and click OK.
11. Repeat Step 4 to select all the worksheet cells.
12. Repeat Step 7 to show the Go To Special dialog box.
13. In the Go To Special dialog box, select the option button for Formulas, leave the Numbers check box selected, and deselect the check boxes for Text, Logicals, and Errors. Click OK.
14. Repeat Step 5 to show the Format Cells dialog box.
15. In the Format Cells dialog box, click the Fill tab, select a color from the palette that is different from the color you selected for Constants in Step 10, and click OK.
16. Select any cell on the worksheet to deselect all the selected special cells.
17. Turn off the Macro Recorder by clicking the Stop Recording button.
18. Before running your new macro to see it in action, repeat Steps 4, 5, and 6 to remove the fill color from all cells.

19. Show the Macro dialog box to run your macro. You can either click the Developer tab on the Ribbon and then click the Macros icon in the Code panel, or you can press the Alt+F8 keyboard shortcut.
20. To run your `My_Macro` macro from the Macro dialog box, select its name in the list box and click the Run button, or double-click its name in the list box.
21. To run your `My_Macro` macro using your keyboard, press the Ctrl+Shift+W shortcut keys you assigned in Step 2.

REFERENCE Please select the video for Lesson 2 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

3

Introducing the Visual Basic Editor

Lesson 2 explains how to create a macro, and you saw a couple of easy ways to run the macro you created. Now it's time to view your macro and have a look at the environment called the *Visual Basic Editor* (VBE), within which all macros and VBA procedures are stored. Seeing where macros live and breathe improves your understanding of the VBA programming process, especially when you start to edit existing macros or create new macros without the Macro Recorder.

WHAT IS THE VBE?

It's fair to say that for many users of Excel, the worksheets, pivot tables, charts, and hundreds of formula functions are all the tools they need to satisfactorily handle their spreadsheet activities. For them, the familiar workbook environment is the only side of Excel they see, and understandably the only side of Excel they are probably aware of.

But Excel has a separate, less visible environment working behind the scenes—the Visual Basic Editor—which is interconnected with the workbook environment even if no programming code exists in the workbook. Both environments are constantly but quietly working together, sharing information back and forth about the entire workbook. The Visual Basic Editor is a user-friendly development environment where programmed instructions are maintained in order to make your spreadsheet applications work.

How to Get Into the VBE

With Excel open, a fast and easy way to get into the Visual Basic Editor is to press Alt+F11 on your keyboard. You can do this from any worksheet. It's just as quick with your mouse, too; you click the Visual Basic Editor icon on the Visual Basic toolbar in versions up to 2003, as shown in Figure 3-1, or the Visual Basic button from the Developer tab on the Ribbon in later versions, as shown in Figure 3-2.



FIGURE 3-1

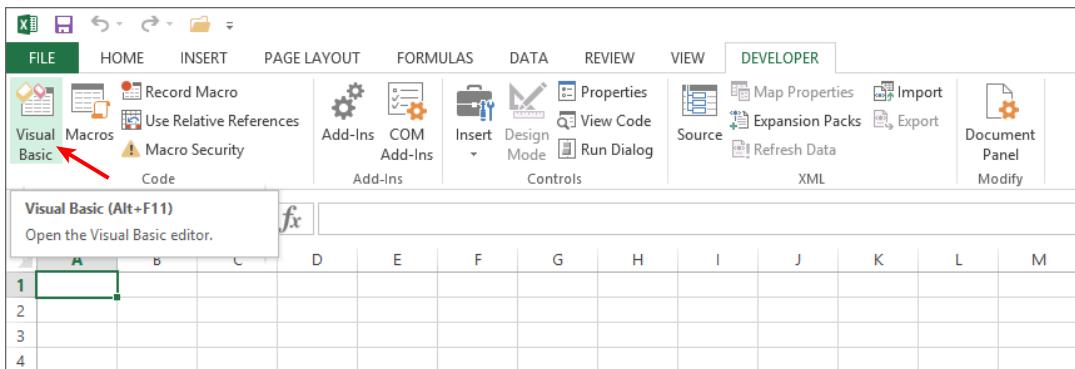


FIGURE 3-2

NOTE If you don't see the Developer tab on your Ribbon, see the steps to show it in Lesson 2, in the section "Accessing the VBA Environment."

CAREFUL, THAT WAS ALT+F11!

The Ctrl key is commonly used in conjunction with other keys for keyboard shortcuts. By force of habit, you might mistakenly press Ctrl+F11 instead of Alt+F11 when attempting to go to the VBE. However, pressing Ctrl+F11 has a curious result: Instead of being taken to the VBE, you will have created an outdated type of sheet called a macro sheet, with the strange tab name of Macro1. Prior to Excel version 97, macros were stored on macro sheets, which you can still create, though they have no practical use with today's Excel, and they no longer hold any programming code. It's OK to just delete the macro sheet if you create one, and take another stab at the Alt key with F11 to get into the VBE.

Understanding the VBE

The Visual Basic Editor can show a number of different windows, depending on what you want to see or do. For the majority of work you do with the help of this book, you want to eventually become familiar with four windows: the Project Explorer window, the Code window, the Properties window, and the Immediate window. Figure 3-3 shows what the VBE looks like with these four windows.

The Project Explorer Window

The Project Explorer is a vertical pane on the left side of the VBE. It behaves similarly to Windows Explorer, with folder icons that expand and collapse when clicked. If you do not see the Project Explorer window in your VBE, press Ctrl+R, or from the VBE menu bar, click View → Project Explorer. As indicated by the first item at the top of the Project Explorer window in Figure 3-3, the name of the workbook I am using (in Excel terms, the VBAProject) is MacroExamples.xlsx.

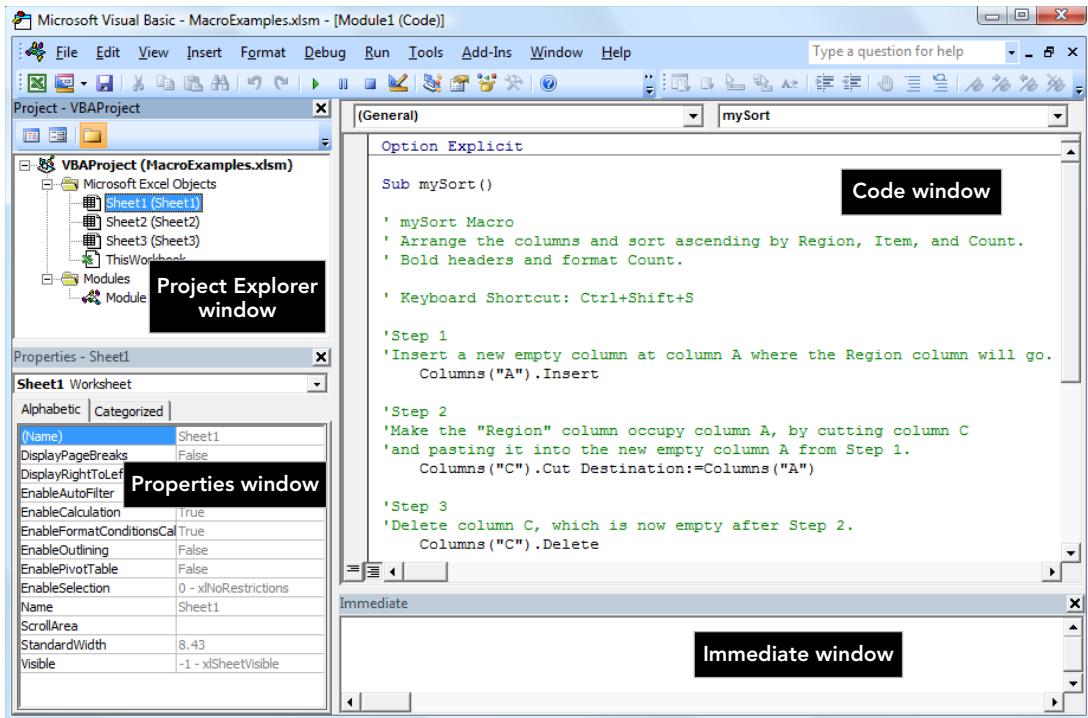


FIGURE 3-3

VBA code is kept in objects known as modules, which are discussed later in further detail. Figure 3-3 shows one module called Module1. Double-clicking a module name in the Project Explorer displays that module's VBA code contents in the Code window, as you see in Figure 3-3.

The Code Window

The Code window is where the code for macros and VBA procedures are located. The VBE provides separate code windows for each module. A good way to think of this is, for every object (worksheet, module, and so on) you see listed in the Project Explorer, the VBE has provided a Code window. Note that the drop-down in the upper right-hand corner of Figure 3-3 displays the name of the macro that is currently showing in the Code window (mySort). As you create multiple macros, you can use this drop-down to quickly move from one macro to another.

The Properties Window

The Properties window is located in the left vertical pane near the bottom of the VBE. If you do not see the Properties window in your VBE, press F4, or from the VBE menu bar click View → Properties Window. This window displays a list of the properties and their assigned values of whatever object is selected in the Project Explorer window. For example, in Figure 3-3, Sheet1 has been selected and the Properties window shows you, among other details, that the Name property for the selected object is Sheet1.

The Immediate Window

The Immediate window is located at the bottom of the VBE, usually below the Code window, as shown in Figure 3-3. If you do not see the Immediate window in your VBE, press Ctrl+G, or from the VBE menu bar click View \Rightarrow Immediate Window. The name “Immediate” has nothing to do with urgency, but rather with the notion that you can query a line of code and immediately obtain its returned result, without having to run a macro to see what that code line does. This comes in handy for code debugging tactics in Lesson 20, but for now I just wanted to point out the Immediate window to familiarize you with its name and location.

Understanding Modules

I touched on modules earlier, but they are worth another mention. A *module* is a container for your code. A single module may hold one or many macros, depending on the workbook and your preference for how you manage your code. For smaller projects with maybe two or three macros, just one module is sufficient. If you develop larger projects with dozens of macros, it’s a good idea to organize them among several modules by theme or purpose.

Several types of modules exist:

- **Standard modules:** These are the kind you have seen already, which hold macros you create from scratch on your own or from the Macro Recorder.
- **UserForm modules:** These belong to a custom user interface object called a UserForm, which is covered in Lessons 21, 22, and 23.
- **Class modules:** These contain the kind of VBA code that enables you to create your own objects programmatically. Creating your own classes is very cool, and you learn about that in Lesson 24.
- **Worksheet modules:** These hold VBA code that looks and acts like macros, but to make things interesting Microsoft refers to that code as a *procedure* instead of as a macro. Worksheet-level procedures are tied to various actions called *events*, such as selecting a range or entering a value in a cell.
- **Workbook module:** Not to be outdone, the workbook itself has its own module, named by default as ThisWorkbook, where code is maintained for handling workbook-level events.

The point is, several types of modules exist, but the concept is the same—modules hold code for the object(s) they serve.

Using the Object Browser

One useful tool the VBE offers is the Object Browser. This section gives some background on the Object Browser and how you can use it to familiarize yourself with locating objects and their associated properties and methods.

The ability to program Excel is based on tapping into any of several libraries of objects in the Microsoft Office objects model. For example, there is an Office library, a VBA library, and of course, an Excel library. Some libraries have hundreds of objects, and each object has many

properties, methods, and, in some cases, associated events. The interwoven collection of object libraries and their keyword kin is enormous. Fortunately, there is the Object Browser to guide your search for information about objects and their properties for whatever library you are interested in.

To see the Object Browser in the VBE, press the F2 key or click View \Rightarrow Object Browser. Figure 3-4 shows the Object Browser—it covers the area normally occupied by the Code window.

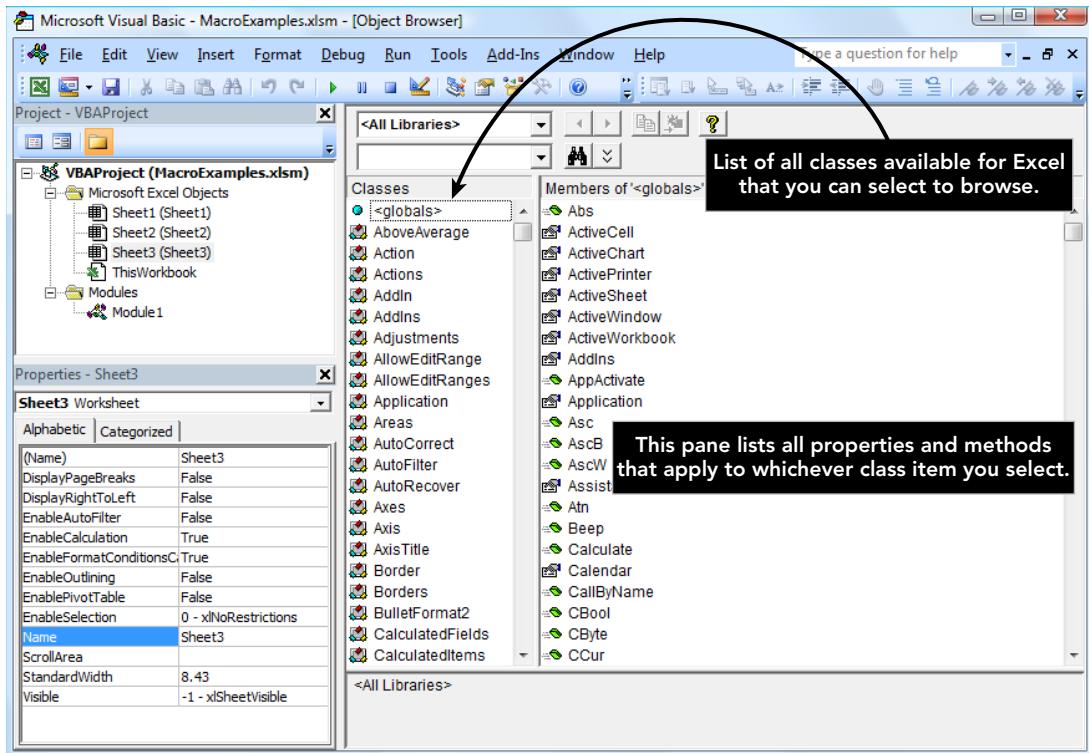


FIGURE 3-4

To get a feel for the Object Browser, click the drop-down arrow next to <All Libraries> and select Excel. When you do that, in the Classes pane you see the classes belonging to Excel. Click the Application class and you see the larger Members pane display the properties and methods relating to the Application object. Click the ActiveWorkbook member and look at the bottom of the Object Browser. You see that ActiveWorkbook is a property that itself is a Workbook object.

After you follow the preceding steps, the Object Browser looks like Figure 3-5; the arrows point to what you clicked. If you click the green Workbook link at the bottom, the Object Browser takes you to the Workbook class and displays the properties and methods for Workbook.

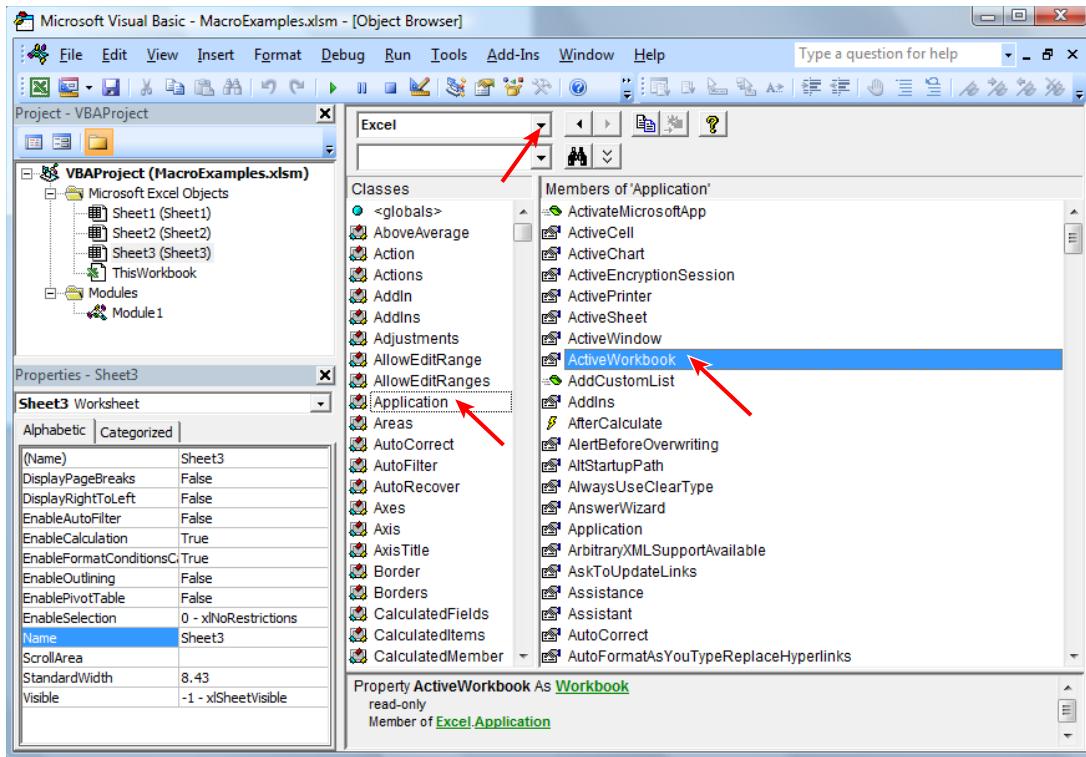


FIGURE 3-5

With a class or member item selected, you can click the yellow question mark icon at the top of the Object Browser to be taken to the Help file for that selected item.

The Object Browser has a Search feature in the drop-down field to the left of the binoculars icon. If you type a term you are interested in and click the binoculars icon, the associated members of that term will be displayed for the selected library.

To exit the Object Browser, click the lower of the two X close buttons near the top-right corner of the VBE.

Exiting the VBE

To exit the VBE and return to the worksheets, you can either press Alt+Q, or click the topmost X close button at the top-right corner of the VBE.

TRY IT

Because this lesson is an introduction to the Visual Basic Editor environment, there are no programming techniques to try, but you can get a jump on your familiarity with the VBE by considering these items:

- You have several ways to get into the VBE, but which way works best for you? As you've seen, Alt+F11 works on all Excel versions, but if you are more of a mouse user than a keyboard user, you have several options depending on what's easiest for you:
 - In version 2003 you can click Tools ⇔ Macro ⇔ Visual Basic Editor, or you can keep the Visual Basic toolbar visible, and click the Visual Basic Editor icon. You can also right-click the workbook icon near the upper-left corner of the Excel window (just to the left of the File menu item), and select View Code, which takes you to that workbook's module in the VBE.
 - In versions 2007, 2010, and 2013, you can click the Visual Basic Editor icon on the Developer tab.
 - In any version of Excel, you can right-click a worksheet tab and select View Code, which takes you to that worksheet's module in the VBE.
- Take another look at the Object Browser and click around its classes and members. The VBA object model is a vast library of information that no one would attempt to memorize, but the idea here is to get a feel for the interwoven relationships among objects' classes, properties, and methods.
- In the Project Explorer window, if you double-click an object such as a worksheet, workbook, or module name, you go directly to that object's Code window. But also notice the pop-up menu when you right-click an object's name in the Project Explorer. Go ahead and click any of those menu items to get the gist of where they lead you and what purpose they serve.
- Get a bit of practice in with the Immediate window. If you were to enter some value into cell A1, and then format cell A1 in bold font, you can enter these expressions in the Immediate window and press Enter for each one:
 - ? Range ("A1") .Value returns whatever value you entered into A1.
 - ? Range ("A1") .Font .Bold returns True if you bolded A1, or False if you did not.
 - ? Range ("A1") .ClearContents returns True and clears the contents of cell A1.

REFERENCE *There is no video or code download to accompany this lesson.*

4

Working in the VBE

In Lesson 3, you took a bird's eye view of the Visual Basic Editor, and you became familiar with the names and locations of its most frequently used windows. In this lesson, you navigate through those VBE windows for the purpose of demonstrating how to handle the kinds of maintenance tasks you will often encounter in the VBE.

TOOLBARS IN THE VBE

The first thing you may have noticed about the VBE interface is that there is no Ribbon. The traditional VBE menu bar is pretty much the same interface for all versions of Excel since 1997.

Because you will be spending more time in the VBE, you'll want convenient access to the toolbar icons relating to the work you'll be doing. If you have not already done so, press Alt+F11 to get into the VBE, and show the Edit and Standard toolbars whose icons will soon come in handy. From the menu bar at the top of the VBE, click View \Rightarrow Toolbars \Rightarrow Edit and again View \Rightarrow Toolbars \Rightarrow Standard, as depicted in Figure 4-1.

MACROS AND MODULES

In Lesson 2, you used the Macro Recorder to create a macro named `mySort`. You learned how to assign a shortcut key to the macro, and how to enter a brief description of what the macro does. You also learned about a couple of ways to run the macro, by using either the shortcut key or the Macro dialog box. One thing you have not been shown yet is the macro itself, or even how to find it.

Locating Your Macros

When the Macro Recorder created the `mySort` macro in Lesson 2, it also created a module in which to store the macro. If this module happens to be the first module of the workbook, as was the case for `mySort`, the Macro Recorder names the new module `Module1` by default. If the Macro Recorder creates another module after that and the workbook still holds a module named `Module1`, the Macro Recorder assigns the default name of `Module2`, and so on.

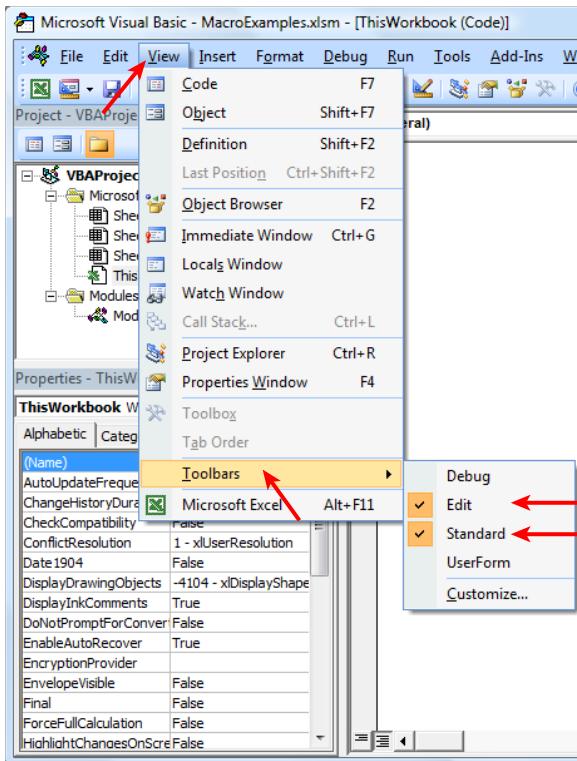


FIGURE 4-1

In the Project Explorer window, expand the bolded VBAProject title (my Project workbook name is MacroExamples.xlsxm) and expand the yellow Modules folder to show the module named Module1. To see the VBA code in that module, you can double-click the module name, or you can right-click the module name and choose View Code, as shown in Figure 4-2.

The mySort macro appears in the Code window for Module1. Based on the steps you took while recording the mySort macro in Lesson 2, Figure 4-3 shows the exact code that was produced by the Macro Recorder in Excel version 2003.

NOTE If you record (or manually compose, as you see in later lessons) a macro in a version of Excel after 2003, and you run that macro in a 2003 version, you might experience an error in that code's execution, depending on what the code is trying to do. VBA code plays well together among versions after 2003, but those later versions of Excel contain newer features, such as Sparklines and an updated object model for charts and pivot tables, that a 2003 version would not recognize. VBA code produced by the Macro Recorder in version 2003 usually works just fine in later versions, but be aware that backward compatibility has its limitations when running code in a 2003 version that was produced in a later version.

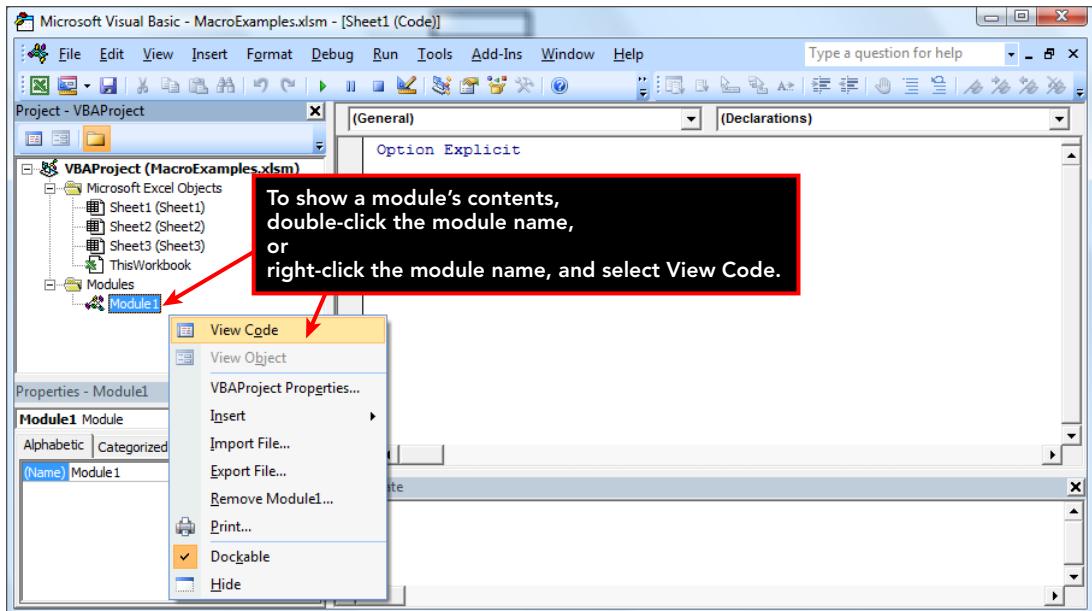


FIGURE 4-2

```

Sub mySort()
'
' mySort Macro
' Arrange the columns and sort ascending by Region, Item, and Count.
' Bold headers and format Count.
'
' Keyboard Shortcut: Ctrl+Shift+S
'

    Columns("A:A").Select
    Selection.Insert Shift:=xlToRight, CopyOrigin:=xlFormatFromLeftOrAbove
    Columns("C:C").Select
    Selection.Cut
    Columns("A:A").Select
    ActiveSheet.Paste
    Columns("C:C").Select
    Selection.Delete Shift:=xlToLeft
    Range("A1:C13").Select
    ActiveWorkbook.Worksheets("Sheet1").Sort.SortFields.Clear
    ActiveWorkbook.Worksheets("Sheet1").Sort.SortFields.Add Key:=Range("A2:A13") -
        , SortOn:=xlSortOnValues, Order:=xlAscending, DataOption:=xlSortNormal
    ActiveWorkbook.Worksheets("Sheet1").Sort.SortFields.Add Key:=Range("B2:B13") -
        , SortOn:=xlSortOnValues, Order:=xlAscending, DataOption:=xlSortNormal
    ActiveWorkbook.Worksheets("Sheet1").Sort.SortFields.Add Key:=Range("C2:C13") -
        , SortOn:=xlSortOnValues, Order:=xlAscending, DataOption:=xlSortNormal
    With ActiveWorkbook.Worksheets("Sheet1").Sort
        .SetRange Range("A1:C13")
        .Header = xlYes
        .MatchCase = False
        .Orientation = xlTopToBottom
        .SortMethod = xlPinYin
        .Apply
    End With
    Range("C2:C13").Select
    Selection.NumberFormat = "#,##0"
    Range("A1:C1").Select
    Selection.Font.Bold = True
End Sub

```

FIGURE 4-3

UNDERSTANDING THE CODE

All macros start with a `Sub` statement (`Sub` is short for `Subroutine`, commonly referred to as a macro) that includes the name of the macro, followed by a pair of parentheses. For the example macro you see in Figures 4-3 and 4-4, the `Sub` statement is simply `Sub mySort()`.

Because this macro was recorded, there is a series of comment lines below the `Sub` statement that the Macro Recorder wants you to know about. For example, you see the macro name, the description of the macro you entered into the Record Macro dialog box, and the notation that the shortcut `Ctrl+Shift+S` has been assigned to this macro.

Comment lines start with an apostrophe, are green in color to help you identify them, and are not executed as VBA code, as opposed to the other lines of VBA code that actually do something when the macro is running.

NOTE *The comments you see in a recorded macro directly reflect the information entered in the Record Macro dialog box. For example, if you assign a shortcut key, or you enter text in the Description field of the Record Macro dialog box as shown in Lesson 2, Figure 2-13, that information will be seen as comments in your recorded macro's code, as shown in Figure 4-3.*

The remaining lines in the macro are VBA statements, and they represent every action that was taken while the Macro Recorder was on:

1. The first thing you did was select column A.
2. Next, you inserted a new column at column A.
3. Next, you selected column C, cut that column, and pasted it to column A.
4. Next, you went back to select column C because it was empty, and you deleted it.
5. Next, you selected range A1:C13 where the table of data was.
6. Next, you sorted the selected range.
7. Next, you selected range C2:C13, which contained numbers you wanted to format.
8. Next, you formatted the selected cells with the thousands comma separator and no decimal places.
9. Next, you selected range A1:C1 where the column labels were.
10. Next, you formatted the selected range in order to Bold the font of those label cells.
11. Finally, you turned off the Macro Recorder, which produced the `End Sub` line. All macros end with the `End Sub` statement.

That's quite a few "Nexts" in the explanation for what is going on! Fortunately, you can edit a macro by typing your own descriptive comments, and you can consolidate a lot of the code so it runs faster and looks cleaner.

EDITING A MACRO WITH COMMENTS AND IMPROVEMENTS TO THE CODE

As good as the Macro Recorder is at teaching VBA code, it is woefully lacking in the efficiency department with the volume of code it produces. To be fair, the Macro Recorder was never meant to be a lean, mean coding machine. Its primary function, which it performs flawlessly, is to produce VBA code that represents your every on-screen action.

It should be said that there is no law in the universe dictating that you must modify your every recorded macro. Sometimes, for simple macros that do the job, leaving them in their original recorded state is fine—if they work the way you want them to, you've won that round.

However, for the majority of VBA code that gets produced by the Macro Recorder, the superfluous and inefficient nature of its excessive code will be impossible to ignore. Besides, when you send your VBA workbook masterpieces to other users, you'll want your code to look and act beyond the beginner stage of recorded code.

NOTE You will find that editing a macro in the Code window is very similar to editing a Word document. Of course, rules exist for proper syntax of VBA code lines, but the principles of typing text, selecting words and deleting them with the Delete key, pressing Enter to go to the next line down—all these word-processor kinds of behaviors with which you are familiar—will help make the macro edit process an intuitive one.

A rule of thumb in VBA development is, don't select or activate objects unless you need to. The methods of Select and Activate are among the biggest culprits of slow, meandering macro execution. For example, the first two lines of code in the recorded macro are:

```
Columns("A:A").Select  
Selection.Insert Shift:=xlToRight
```

Those two lines can and should be consolidated into one line, bypassing the Selection activity:

```
Columns("A").Insert Shift:=xlToRight
```

Same with the next two statements:

```
Columns("C:C").Select  
Selection.Cut Destination:=Columns("A:A")
```

which can be expressed more succinctly as:

```
Columns("C").Cut Destination:=Columns("A")
```

You can see where I am going with this. In VBA, you can act directly upon most objects, most of the time, without needing to select them. When you deleted column C, you never needed to touch it in order for VBA to do the work for you, because the following statement:

```
Columns("C:C").Select  
Selection.Delete Shift:=xlToLeft
```

can become this:

```
Columns("C").Delete Shift:=xlToLeft
```

Figure 4-4 shows how the original 13 lines of code in the mySort macro have been reduced to a much more readable and highly efficient six lines. Also notice how comments can be added for the purpose of enhancing the organized look of the macro. Your comments will help you, and anyone reading the macro, to understand what the code lines are doing, and why they are doing it.

```
Sub mySort()

' mySort Macro
' Arrange the columns and sort ascending by Region, Item, and Count.
' Bold headers and format Count.

' Keyboard Shortcut: Ctrl+Shift+S

'Step 1
'Insert a new empty column at column A where the Region column will go.
    Columns("A").Insert

'Step 2
'Make the "Region" column occupy column A, by cutting column C
'and pasting it into the new empty column A from Step 1.
    Columns("C").Cut Destination:=Columns("A")

'Step 3
'Delete column C, which is now empty after Step 2.
    Columns("C").Delete

'Step 4
'Sort range A1:C13 by column A ("Region"), column B ("Item"), and column C ("Count").
    Range("A1:C13").Sort Key1:=Range("A2"), Order1:=xlAscending, Key2:=Range("B2"),
        Order2:=xlAscending, Key3:=Range("C2"), Order3:=xlAscending, Header:=xlYes

'Step 5
'Format the numbers in the Count column to show the thousands comma separator.
    Range("C2:C13").NumberFormat = "#,##0"

'Step 6
'Bold the header labels of "Region", "Item", and "Count" in range A1:C1.
    Range("A1:C1").Font.Bold = True
End Sub
```

FIGURE 4-4

NOTE You've now seen plenty of comments in the example macros, and how useful comments can be in your VBA code. To enter a comment line of text, simply type the apostrophe character, and everything you type after that, on that same line, will be regarded as a comment and not executed as VBA code. Usually, comments are written as standalone lines of text, meaning the very first character on that line is the apostrophe. However, some programmers prefer to place comments on the same line as actual VBA code. For example:

```
Range("A1").Clear 'Make cell A1 be empty for the next user.
```

In any case, comments will be green in color by default, and will not be executed as VBA code.

Another way you can speed up your macros is to use the `With` statement when you are performing multiple actions to the same object, such as to a range of cells. Suppose as part of your macro you need to clear a range of cells and format the range for the next user. If you use the Macro Recorder to do this, here is the code you might get:

```
Range("A1:D8").Select
Selection.Clear
Selection.Locked = False
Selection.FormulaHidden = False
Selection.Font.Bold = True
Selection.Font.Italic = True
```

Notice there are five lines of code that all start with the `Selection` object, which refers to the selected range of A1:D8. If this code were to run as the Macro Recorder produced it, VBA would need to resolve the `Selection` object for each line of code.

You can do two key edits to these lines of code by avoiding the `Select` method altogether and referring to the range object only once at the beginning of a `With` structure. Between the `With` and `End With` statements, every line of code that starts with a dot is evaluated by VBA as belonging to the same range object, meaning the range reference need only be resolved once. Here is the condensed code using a `With` structure for greater efficiency:

```
With Range("A1:D8")
    .Clear
    .Locked = False
    .FormulaHidden = False
    .Font.Bold = True
    .Font.Italic = True
End With
```

Deleting a Macro

There will be many times when you have recorded or composed a macro that you don't need any more. Instead of having a useless macro hanging around doing no good, it's better to delete it. To delete a macro, you can select its entire code in the Code window (be sure you only select from and including the `Sub` line to and including the `End Sub` line) and press the Delete key.

NOTE You can delete a macro from outside the VBE. While on any worksheet, if you press Alt+F8 to call the Macro dialog box, you can select the macro name in the list and click the Delete button.

Inserting a Module

With larger VBA projects, you'll want to distribute your macros among two or more modules. With large projects, you'll be organizing your macros by some kind of theme or purpose. For example, the macros in your company's budget workbook that deal with reports might be placed in their own module. Sometimes you will have no choice in the matter, because modules do have a limit as to how much code they can individually support. To insert a new module, from the VBE menu bar, select `Insert` → `Module`, as shown in Figure 4-5.

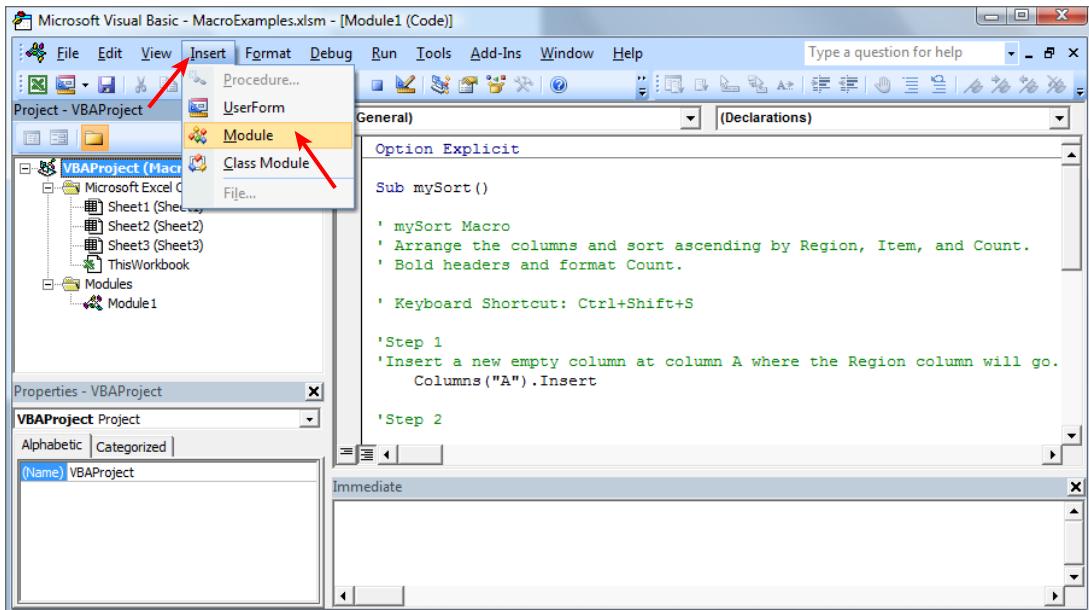


FIGURE 4-5

You'll see that your new module appears in the Project Explorer window. The entry cursor will be blinking in the new Code window, all primed and ready for you to enter VBA code into your new module, as depicted in Figure 4-6.

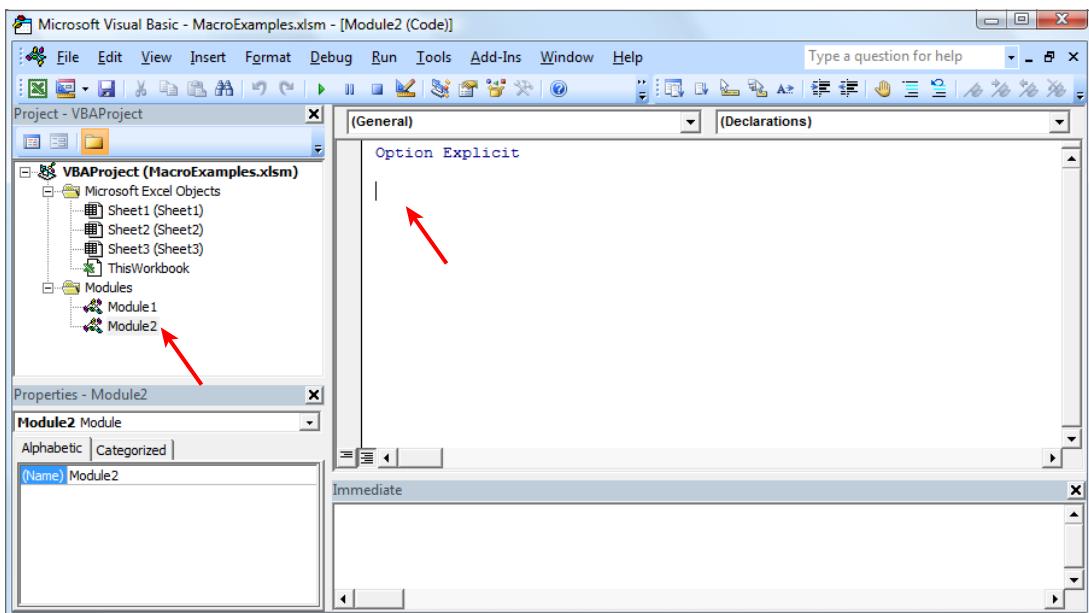


FIGURE 4-6

Renaming a Module

You've noticed that the Macro Recorder assigned the default name of Module1 to the module it created, and just now with Module2 you see how Excel continues to assign a sequential default name to subsequent modules you insert. Yep, definitely a pattern going on here with the module names, but it doesn't mean those names need to stay that way.

You can change a module name, and it makes a lot of sense to do so. This is especially true when you have a complex workbook containing many macros that are organized in several modules, and you want the module names to describe the themes of the macros they contain.

To change a module name, select it by clicking its original name in the Project Explorer. Notice in the Properties window that the Name property of the selected module object is, as you would expect, Module2. In the Properties window, use your mouse to select the entire module name property, such as you see in Figure 4-7.

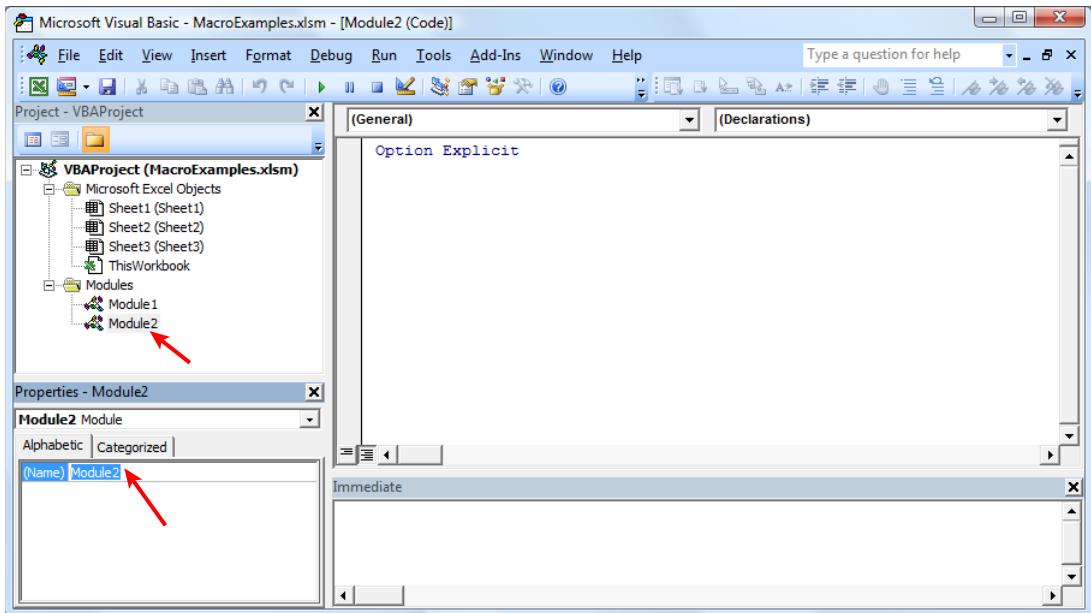


FIGURE 4-7

Now, it's a simple task of typing over the selected Module2 text in the Properties window as you enter whatever new name you want to give to that module. For this demonstration, name the module Test. Just type the word **test** and press Enter. The successful result is shown in Figure 4-8.

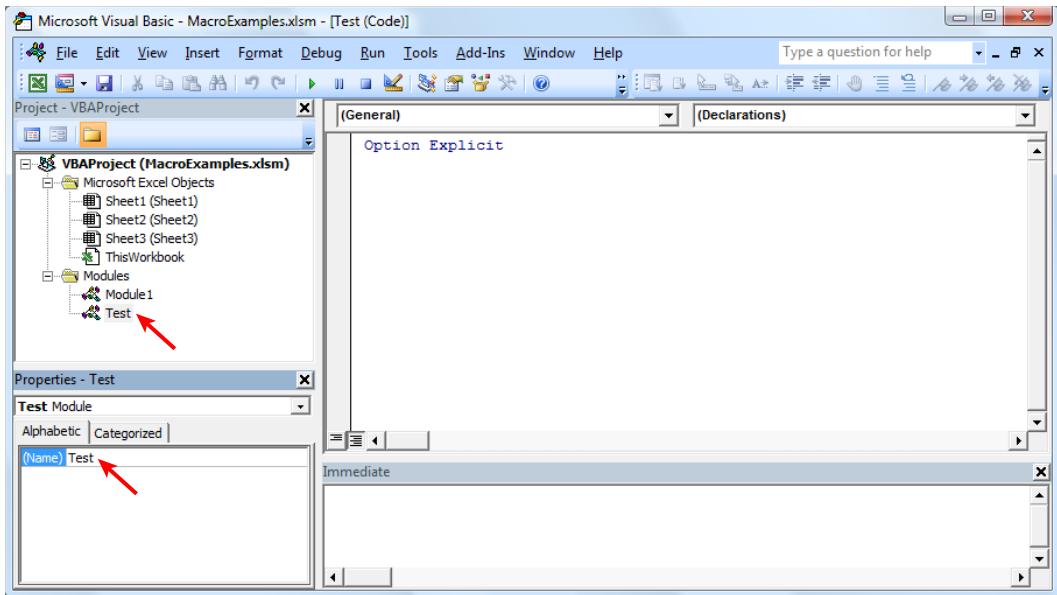


FIGURE 4-8

Deleting a Module

You can delete an entire module, and it's wise to keep your projects uncluttered of unused module objects if they have served their purpose and will no longer hold any macros. To delete a module, right-click the module name in the Project Explorer, and from the pop-up menu, click Remove [module name], as shown in Figure 4-9.

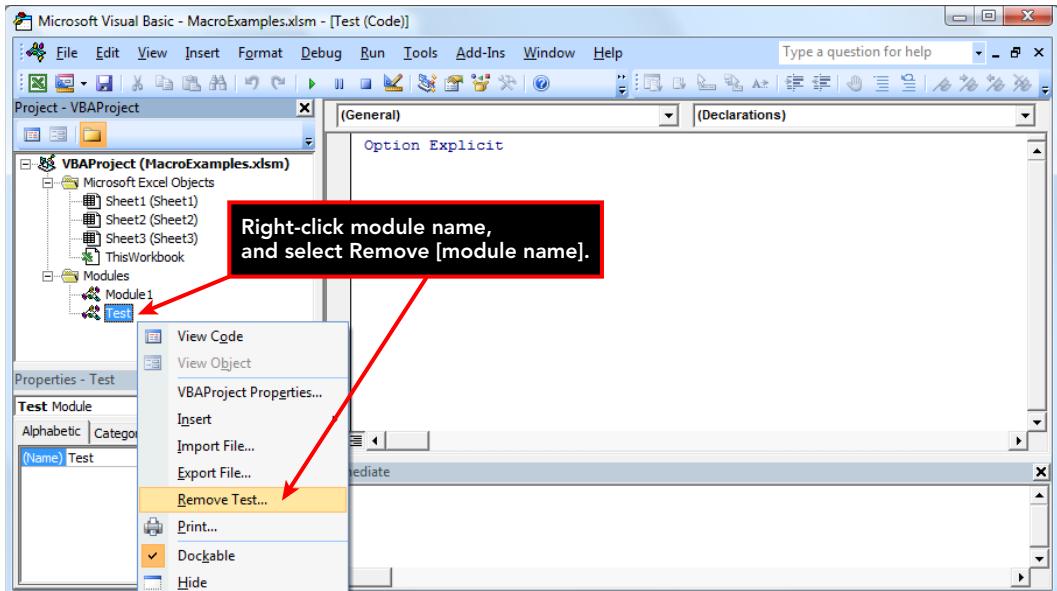


FIGURE 4-9

You're prompted with a message to confirm your intentions, along with a question as to whether you want to export your module elsewhere. In very remote instances you will need to export a module, but I have never come across a need to do that. Although the default button on the message is Yes, you will usually click the No button, as shown in Figure 4-10, to confirm the deletion of that module.

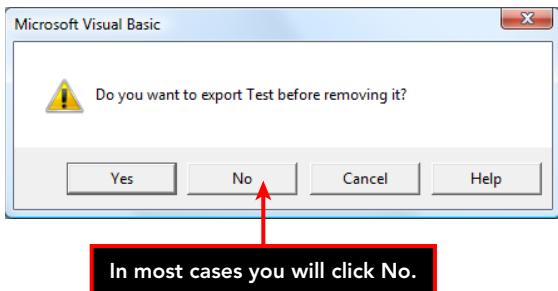


FIGURE 4-10

Locking and Protecting the VBE

The beauty of macros is that when they are properly constructed, you can count on them to do their job. The last thing you want is for another user of your workbook to wander into the Visual Basic Editor and make any kind of keystroke in a Code window. Especially when other people are using your workbook, you will want to protect your code from uninvited guests.

To limit access to the VBE, click Tools \Rightarrow VBAProject Properties, which calls the VBAProject - Project Properties dialog box. Click to select the Protection tab. Place a checkmark in the box next to Lock Project for Viewing. Enter a password you will remember, and confirm it, as shown in Figure 4-11.

Click OK to exit the dialog box. For the locked protection to take effect, you need to save the workbook and close it. Now, each time the workbook is reopened, the Visual Basic Editor will require your password if you or anyone tries to gain access to the VBE.

WARNING Excel passwords are case sensitive. If your password attempt to access a locked VBE is rejected, the reason might be due to an incorrect upper- or lowercase entry.

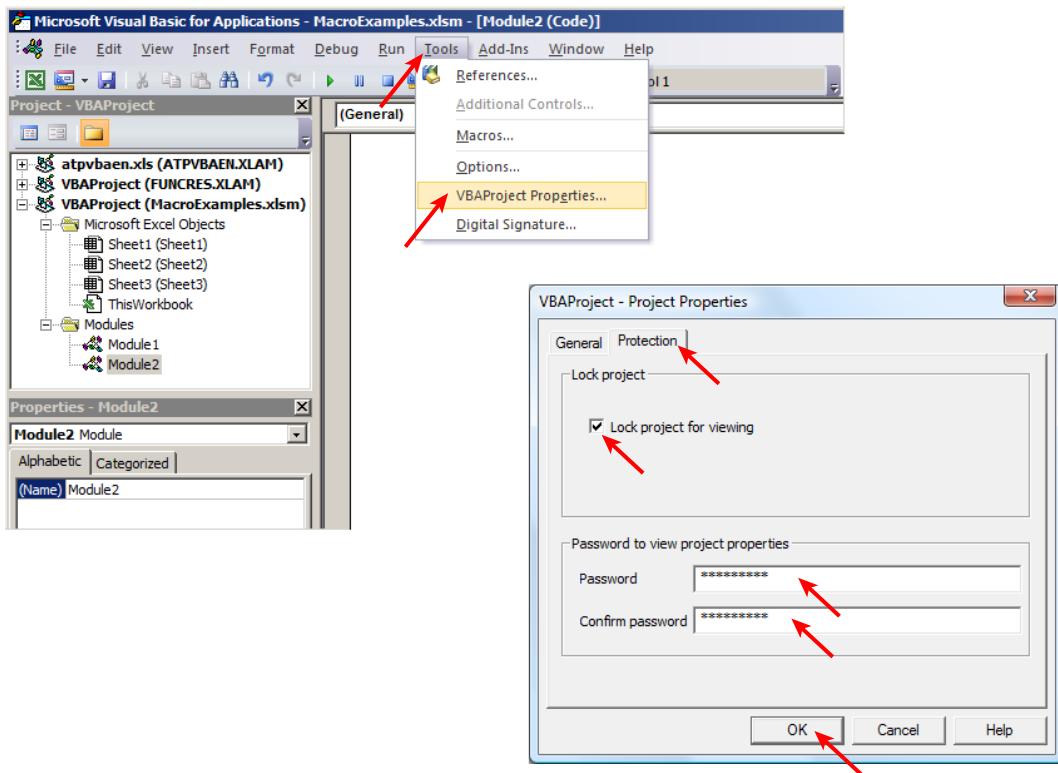


FIGURE 4-11

TRY IT

In this lesson, you practice inserting a new module into the VBE, and pasting a macro that you copy from a website into the new module.

Lesson Requirements

For this lesson, you need access to the Internet.

To get the sample workbook file, you can download Lesson 4 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

In Step 2, and in the video that accompanies this lesson, I use my website as the source for the macro that gets copied. There are many excellent websites with more VBA examples, some of which I list on my Links page at www.atlaspm.com/excel/#Links.

Step-by-Step

Place a macro from an external source into a new Excel module. In this exercise, a website is being used as the external source of the macro you want to copy and put into your workbook.

1. Open Excel, and open the workbook that will hold the macro you'll be importing.
2. Open your web browser and go to the website holding the macro you want to copy. In this example, my website at www.atlaspm.com is the source for the macro being copied for import. Enter the search keyword(s) in the Search field to reveal the link(s) that show a macro example that handles the task you want to solve.
3. Copy the macro from that source page onto your clipboard.
4. Return to your workbook.
5. Go to the Visual Basic Editor by pressing your keyboard's Alt+F11 keys, or by clicking the Developer tab on the Ribbon and selecting the Visual Basic icon.
6. From the menu bar in the VBE, click Insert \leftrightarrow Module
7. In your new module, you can paste the macro you copied in Step 3 by pressing Ctrl+V on your keyboard, or you can right-click anywhere in your new module and select Paste.
8. Return to your worksheet by pressing the Alt+Q keys or by clicking the Close button in the top-right corner of the VBE.
9. To run your macro from the Macro dialog box, press the Alt+F8 keys or click the Macros icon on the Developer tab.

REFERENCE Please select the video for Lesson 4 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

PART II

Diving Deeper into VBA

- ▶ **LESSON 5:** Object-Oriented Programming: An Overview
- ▶ **LESSON 6:** Variables, Data Types, and Constants
- ▶ **LESSON 7:** Understanding Objects and Collections
- ▶ **LESSON 8:** Working with Ranges
- ▶ **LESSON 9:** Making Decisions with VBA

5

Object-Oriented Programming: An Overview

In Lesson 1, you saw a brief historical synopsis of VBA. One particular facet of VBA’s evolution that is worth more explanation is object-oriented programming, or OOP.

Object-oriented programming came about in the 1980s as a new concept in computer programming. Its popularity grew over time and with good reason—OOP’s original precepts are at the core of today’s VBA programming language for Excel.

WHAT “OBJECT-ORIENTED PROGRAMMING” MEANS

Visual Basic for Applications is an object-oriented programming language. The basic concept of object-oriented programming is that a software application (Excel in this case) consists of various individual objects, each of which has its own set of features and uses. An Excel application contains cells, worksheets, charts, pivot tables, drawing shapes—the list of Excel’s objects is seemingly endless. Each object has its own set of features, which are called properties, and its own set of uses, called methods.

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

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

Therefore, we can say that object-oriented programming, upon which VBA is based, is a style of programming language that cares primarily about objects, and how those objects can be manipulated based on their intrinsic qualities.

THE OBJECT MODEL

The Excel object model is the heart and soul of how VBA is used in Excel. Although VBA is the programming language for Excel, it is also the programming language for Office applications in Word, Access, PowerPoint, and Outlook. Even though all these applications are programmable with VBA, they have their own programming needs because they are different software applications, and hence are designed to serve different functions. Excel does not receive e-mails as Outlook does, and Word does not produce reports from its own database tables as Access does.

Every VBA action you take in your Excel workbook sends a command through the Excel object model. The object model is a large list of objects that relate to Excel, such as worksheets, cells, ranges, and charts. The VBA code in your macro that adds a worksheet to the workbook will make sense to Excel because it is communicating with the objects that are recognized to be present in the Excel object model. For example, that same macro to add a worksheet would not work in Outlook. The Outlook object model does not include worksheets, because Outlook is an application that maintains e-mails and appointment calendars, not worksheets.

The object model of any VBA application is hierarchical by design. In the Excel object model, the `Application` object is at the top of the model because it is the entire Excel application. Under the `Application` object is a whole host of other objects, one of them being the `Workbook` object. Under `Workbook` is the `Worksheet` object, among many others, and under the `Worksheet` object are `Range` and `Cell` objects, and so on.

The result of this hierarchy is what drives the proper syntax for your VBA macros. For example, if you want to enter the word “Hello” in cell A1 of Sheet1 of the workbook you are currently working in, the line of code to handle that could be the following:

```
Application.ActiveWorkbook.Worksheets("Sheet1").Range("A1").Value = "Hello"
```

VBA is a smart language. It knows you are working in Excel if you are specifying a `Workbook` object. It also knows you are doing something in a workbook if you are specifying a `Worksheet` object. Therefore, the preceding line of code can be shortened to this:

```
Worksheets("Sheet1").Range("A1").Value = "Hello"
```

That can be shortened further if you are working on Sheet1 (that is, if Sheet1 is the active sheet) when the code line is executed. If the parent `Worksheet` object is not specified, VBA’s default assumption is that you want the active worksheet to receive the word “Hello” in cell A1, and in that scenario the line of code would simply be this:

```
Range("A1").Value = "Hello"
```

A bit of theory on the subject of objects. In an object-oriented programming environment, VBA regards as an Excel object pretty much any element of the Excel application you can think of, whether it is a button, or a row, or a window—even the Excel application itself.

When you add an object to your workbook with VBA—for example, if you run a macro that creates a chart—VBA is at work behind the scenes, storing information about that `Chart` object, and assigning default values to its properties that were not specified in the macro. I mention this as a piece of good news, because with VBA filling in the blanks as it does, it’s that much less about VBA

you need to learn to start writing advanced macros. This advantage will become clearer as you progress into more complex programming techniques.

Properties

As noted earlier, VBA objects have inherent qualities, called *properties*, similar to any objects you may deal with in the real world. Properties define what the object looks like and how it acts. If you own a red bicycle, you can change its `Color` property by painting the bicycle a different color. For a `Cell` object on a worksheet, you can change its `Color` property by formatting the cell with a different fill color.

In VBA code, you refer to the property of an object by first referring to the object, then the property, separated by a dot. Following are examples of a few of the many properties belonging to the `Cell`, `Worksheet`, and `Workbook` objects.

This line of code would format the active cell's `Locked` property:

```
ActiveCell.Locked = True
```

The `Name` property of the `Worksheet` object represents the worksheet's tab name. For example, this expression in the Immediate window would return the name of the active worksheet:

```
? ActiveSheet.Name
```

This expression would change the `Name` property of the active worksheet to "Hello", and when executed would result in "Hello" being the active worksheet's new tab name:

```
ActiveSheet.Name = "Hello"
```

The following expression will change the `Color` property of the active worksheet's tab to yellow:

```
ActiveSheet.Tab.Color = vbYellow
```

Workbooks have a `Saved` property that indicates if the workbook has been saved since its most recent change. For example, if you save your workbook and then enter the following expression in the Immediate window, VBA will return `True`:

```
? ThisWorkbook.Saved
```

If you were to make some change to the workbook, such as entering a number in a cell, and immediately re-evaluate the expression `? ThisWorkbook.Saved`, `False` would be returned because VBA knows that the workbook has not been saved since it was last changed.

Methods

Methods are actions that can be performed by objects. VBA objects have inherent behavioral abilities. Following are examples of Excel objects and some of their methods.

The `Range` object of A1:D10 can have its cells' contents cleared with the `ClearContents` method:

```
Range("A1:D10").ClearContents
```

Workbooks and worksheets can be activated with the `Activate` method:

```
Workbooks("Book1.xlsx").Activate  
Worksheets("Sheet2").Activate
```

Here's a more complicated example, to call your attention to the fact that objects can contain objects, not just properties. Suppose you have three pivot tables on Sheet1, and you only want to refresh the pivot table named PivotTable2. As far as VBA is concerned, what you really want to refresh is the `PivotCache` object of the `PivotTable2` object of the `Sheet1` worksheet object. This line of code would accomplish that, using the `Refresh` method:

```
Worksheets("Sheet1").PivotTables("PivotTable2").PivotCache.Refresh
```

NOTE *This multiple-object syntax might look daunting at first, but you can take some comfort in knowing that you've been writing VBA code in this manner since Day 1. All objects (except the Application object, which is Excel itself) have a Parent property—that is, another object to which they belong. In many cases, you don't need to specify the Parent object because it is inferred by default. For example, if you are referring to cell A1 on your active worksheet, you do not need to (though you could) express it as ActiveSheet .Range ("A1")—you only need to express it as Range ("A1"). In the preceding example, however, pivot tables are embedded objects for which VBA requires you to specify the parent worksheet object. If all this talk of properties and methods is not clear yet, don't worry, it will all make perfect sense when you see the theory in action.*

Collections

Some of the VBA programming you learn in later lessons involves the concept of *collections*, and it is a topic I'm touching on here. In object-oriented programming, a `Collection` is an object that contains a group of like objects. For example, there is a `Worksheets` collection object that is the entire group of `Worksheet` objects in your workbook. Even if one worksheet contains hundreds of formulas and another worksheet is totally empty, both those worksheets are like objects because they are both worksheets, and therefore they both are a part of the `Worksheets` collection.

As you'll see, invoking the `Collection` object in your code is a terrific way to take some action on all the objects in that collection, without needing to know anything specific about the collected objects. For example, say you want to add some boilerplate text to every comment on your worksheet. Employing a `For . . . Each` loop (loops are covered in Lesson 10) to edit every comment in the `Comments` collection would make the task simple because each comment would belong to the `Comments` collection, and you'd be confident knowing you hit all comments without needing to know what cells they are in.

NOTE A good rule of thumb in recognizing a Collections object is to notice that its name ends with the letter *s*, as a pluralized form of its singular object item name. Examples of this are the Names collection of individual Name objects, the Charts collection of individual Chart objects, the Workbooks collection of individual Workbook objects, and so on.

TRY IT

This lesson provided an overview of object-oriented programming. There are no programming techniques to try based on the material in this lesson, but here are some important concepts to keep in mind:

1. Excel is replete with objects, such as workbooks, worksheets, and cells, and each object has its own set of properties that can be altered to suit your application project's design.
2. If you should need to refer to an object's container, such as when you refer to a worksheet in another workbook, just use the object's Parent property. All objects (except Application) have a Parent property that is the object within which they are contained. For example, if your active workbook object is Book2 but you want to refer to Sheet1 in Book1, you'd precede the Sheet1 object with its parent Book1 object name, like this:

```
Workbooks ("Book1.xlsxm") .Worksheets ("Sheet1") .Range ("A1") .Value = "Hello"
```

3. The Application object indeed holds the highest order of Excel's objects, but as you will see, it also offers many useful methods and properties. The Application object provides the ability to insert worksheet functions (SUM, AVERAGE, VLOOKUP, and so on), as well as commands to control Excel's display options for worksheet gridlines, tabs, and window sizes.

REFERENCE There is no video or code download to accompany this lesson.

6

Variables, Data Types, and Constants

Many of the macros you develop will involve the need for referencing an item you are working on without specifying that item by its name, amount, or location. This concept may sound strange at first, but you will quickly discover with your macros that in many situations it makes sense, and indeed is necessary, to manipulate or analyze data in one part of your macro, and hold the results in virtual memory for later use.

WHAT IS A VARIABLE?

VBA stores data in memory using a variable. A *variable* is a name given by you, to which you assign a piece of data that is stored in an area of the computer's memory, allowing you to refer to that data when you need to later in the macro. VBA handles the task of finding an appropriate place in the computer's memory to store your variable data, and dutifully retrieves the data when you ask for it by its variable name.

Variables hold values of different data types (more on this later) that are specified when the variable is declared. When you declare a variable, you do so by entering a declaration statement that includes four keywords in a particular order:

1. The `Dim` statement (VBA's abbreviation for Dimension), which all variable declarations start with.
2. The name of your variable, which you create, such as `myValue`.
3. The word `As`.
4. The type of data being stored.

One common data type is called `Integer`, which, as you see in Table 6-1, refers to whole numbers within a certain range. Using the preceding four steps as a sequential construction guide, here is a typical-looking variable declaration statement:

```
Dim myValue As Integer
```

NOTE *A few rules in VBA for variable names:*

- Cannot be greater than 255 characters in length.
- Cannot contain a space.
- Cannot contain mathematical operation characters +, -, /, *, =, <, >, or ^.
- Cannot contain punctuation characters, such as a comma, period, question mark, or exclamation.
- Cannot contain characters @, #, \$, %, &-, (,), {, }, [,], \, :, “, ‘, ’, ~, or |.
- Cannot be terms reserved in VBA, for example Dim, Sub, or Function.
- Must be unique in the macro or procedure that uses it.
- May contain, but cannot start with, a number or an underscore character.

Basically, when it comes to naming your variables, keep it simple. Use only letters (and maybe numbers after the first character) for a name that is concise and gives a clue as to the general purpose of the variable.

You'll soon see the enormous benefit that this kind of innocent-looking statement can have in your macro. Although a few wrinkles exist in the variable declaration process, a variable declaration statement will often look no more complicated than this.

ASSIGNING VALUES TO VARIABLES

After the variable declaration statement, which might be the next code line or 100 code lines later in your macro, depending on what you are doing, you will have a statement that assigns some value or attribute to the myValue variable. Here's an example of assigning the number in cell A1 to the myValue variable:

```
myValue = Range("A1").Value
```

The value you assign might be an actual value that is stored in a cell, as in the preceding example, or it might be a value you create or define in some way, again, depending on the task at hand. This notion will become clearer with more examples you'll be seeing throughout the book.

WHY YOU NEED VARIABLES

I mentioned earlier that in some situations, employing a variable will be a sensible option. Suppose you have a number in cell A1 that you are referring to for several analytical purposes throughout your macro. You could retrieve that number by referring to its A1 cell address every time, but that would force Excel to look for the same cell address and to recommit the same number to memory every time.

As a simplified example, here is a macro with four commands, all invoking the value in cell A1:

```
Sub WithoutVariable()  
Range("C3").Value = Range("A1").Value  
Range("D5").Value = Range("A1").Value / 12
```

```

Range("E7").Value = Range("A1").Value * 365
MsgBox "The original value is " & Range("A1").Value
End Sub

```

For VBA to execute this macro, it must go through the same behind-the-scenes gyrations four separate times to satisfy each of the four commands that reference range A1. And if your workbook design changes, where you move the number of interest from cell A1 to cell K5, you need to go into the code, find each related code line, and change the cell reference from A1 to K5.

Fortunately, there is a better way to handle this kind of situation—by declaring a variable to refer to the value in cell A1 just once, like this:

```

Sub WithVariable()
Dim myValue As Integer
myValue = Range("A1").Value
Range("C3").Value = myValue
Range("D5").Value = myValue / 12
Range("E7").Value = myValue * 365
MsgBox "The original value is " & myValue
End Sub

```

By assigning the number value in cell A1 to the `myValue` variable, you've increased your code's efficiency and its readability, and VBA will keep the number value in memory without having to reevaluate cell A1. Also, if your cell of interest changes from A1 to some other cell, say cell K5, you only need to edit the cell address in the assignment code line to refer to cell K5, like so:

```
myValue = Range("K5").Value
```

As you've probably noticed in this situational example, a variable declaration is advisable, but it is not an absolute requirement for the `WithoutVariable` macro to function. However, as you will see in the upcoming lessons, variable declaration will be a necessary practice for handling more complex tasks that involve loops, object manipulation, and conditional decision-making. Don't worry—after you see a few examples of variables in action and start practicing with them on your own, you'll quickly get the hang of when and how to declare variables.

DATA TYPES

Simply stated, VBA's role in life is to manipulate data in a way your computer can understand it. A computer sees information only as a series of binary numbers such as 0s and 1s—very differently than how humans see information as numerals, symbols, and letters of the alphabet.

Your macros will inevitably manipulate data of varying types, such as text, numbers, or Range objects. Part of VBA's job is to bridge the communication gap between humans and computers, by providing a method for telling the computer what type of data is being referred to in code. When you specify a data type in VBA, you help the computer to know how it should regard your data so that your macros will produce the results you'd expect, based on the types of data you are manipulating.

Understanding the Different Data Types

Data types are the different kinds of ways you can store data in memory. Table 6-1 shows a list of common data types with their descriptions and memory usage.

TABLE 6-1: Data Types

DATA TYPE	DESCRIPTION	MEMORY
Boolean	True or False; 1 or 0; On or Off.	2 bytes
Byte	An integer from 0 to 255.	1 byte
Currency	A positive or negative number with up to 15 digits to the left of the decimal point and up to 4 digits to the right of it.	8 bytes
Date	A floating-point number with the date to the left of the decimal point and the time to the right of it.	8 bytes
Decimal	An unsigned integer scaled to the power of 10. The power of 10 scaling factor specifies the number of digits to the right of the decimal point, and ranges from 0 to 28.	12 bytes
Double	A floating-point number ranging in value from -1.79769313486231E308 to -4.94065645841247E-324 for negative values and from 4.94065645841247E-324 to 1.79769313486232E308 for positive values.	8 bytes
Integer	An integer ranging from -32,768 to 32,767.	2 bytes
Long	An integer ranging from -2,147,483,648 to 2,147,483,647.	4 bytes
Object	A reference to an object, such as a range of cells, a chart, a pivot table, a workbook, a worksheet, or any one of the many other objects that are a part of the Excel application.	4 bytes
Single	A floating-point number ranging in value from -3.402823E38 to -1.401298E-45 for negative values and from 1.401298E-45 to 3.402823E38 for positive values.	4 bytes
String	There are two kinds of strings: variable-length and fixed-length. A variable-length string can contain up to approximately 2 billion characters. A fixed-length string can contain 1 to approximately 64,000 characters.	For a variable-length string, 10 bytes plus storage for the string. For a fixed-length string, the storage for the string.
Variant	Data type for all variables that are not explicitly declared as some other type, which can contain any kind of data except fixed-length string data.	For containing numbers, 16 bytes. For containing characters, 22 bytes plus storage for the characters.

Declaring a Variable for Dates and Times

The `Date` data type is worth an extra look because it is the data type with which variables for both dates and times can be declared. You can assign values to a date variable by enclosing them in the # (number sign) character, with the value being recognizable to Excel as either a date or time. For example:

```

myDate = #09 October 1958#
or
myDate = #October 9, 1958#
or
myTime = #9:10 PM#
or
myTime = #10/9/1958 9:10:00 PM#

```

NOTE When entering dates, get into the good habit of entering the year as a full four-digit number. The year 2029 is the dividing line in VBA for two-digit years belonging to either the twentieth or twenty-first centuries. All two-digit years from 00 to and including 29 are regarded as belonging to the 2000s, and 30 to 99 are regarded as belonging to the 1900s. For example, the expression 10/10/29 in Excel is October 10, 2029, but 10/10/30 is regarded by Excel as October 10, 1930.

Declaring a Variable with the Proper Data Type

As you become more familiar with VBA, you'll notice that different developers have their preferred writing styles when declaring variables. For example, you can declare several variables on one line, each separated by a comma, like this:

```
Dim myValue1 as Integer, myValue2 as Integer, myValue3 as Integer
```

There is nothing wrong with that construction, but be careful not to make this common mistake:

```
Dim myValue1, myValue2, myValue3 as Integer
```

If you do not specify a data type after a variable name, such as in the latter case with `myValue1` and `myValue2`, VBA assigns the default variant data type. Only the `value3` variable has been specified the `Integer` data type. Variant is a catch-all data type that is the most memory-intensive, and the least helpful in understanding the purpose of its associated variables if anyone else should read your code.

The Variant data type does have its place, for instance when dealing with arrays or conversions of data types, but you should take care to specify the appropriate data types of all your variables. In so doing, your macros will run faster, they'll be easier to read, and they'll be more reliable.

FORCING VARIABLE DECLARATION

Declaring your variables can only be a good thing. It takes a little extra thought and effort, but not declaring your variables can cause a lot more trouble when reading or debugging your code. Macros run faster and use less memory when all variables are properly declared.

You can tell if variable declaration is being enforced by seeing if the statement `Option Explicit` is at the top of your module. If you do see the `Option Explicit` statement, write a quick macro that tries to call an undeclared variable, such as you see depicted in Figure 6-1. When you attempt to

run the macro, you receive a compile error as shown in Figure 6-1, informing you a variable is not defined. In this scenario, the error occurred because the myName variable was not declared with a statement such as Dim myName as String.

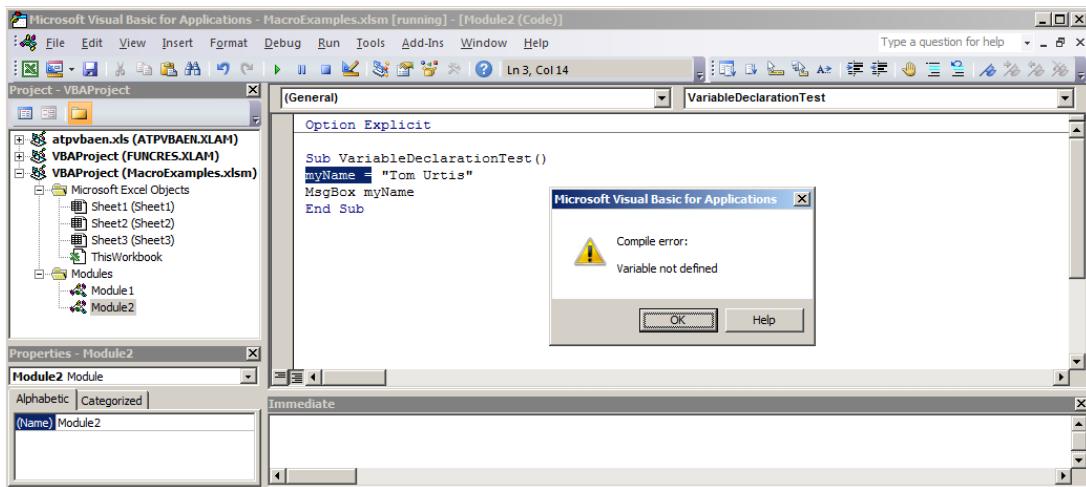


FIGURE 6-1

If you do not see the Option Explicit statement at the top of your modules, go into the VBE and from the menu bar, select Tools ➔ Options, as shown in Figure 6-2.

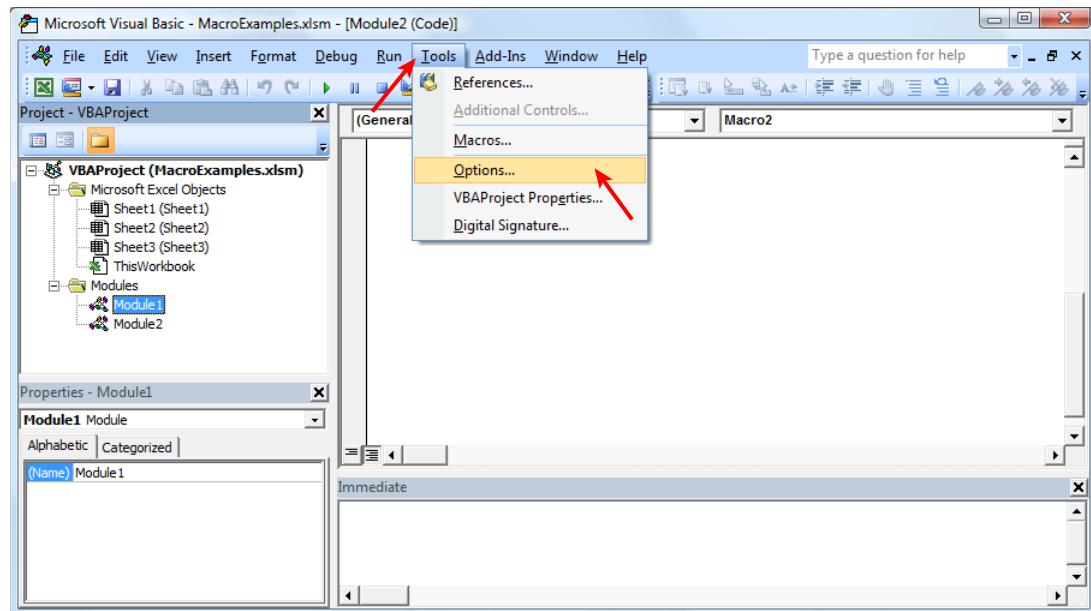


FIGURE 6-2

You see the Options dialog box. On the Editor tab, select the option Require Variable Declaration, as shown in Figure 6-3, and click OK.

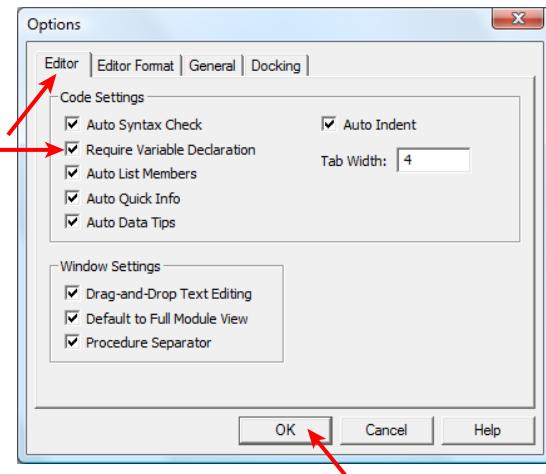


FIGURE 6-3

Figure 6-4 shows the Option Explicit statement at the top of the module. The statement appears in every new module you insert thereafter.

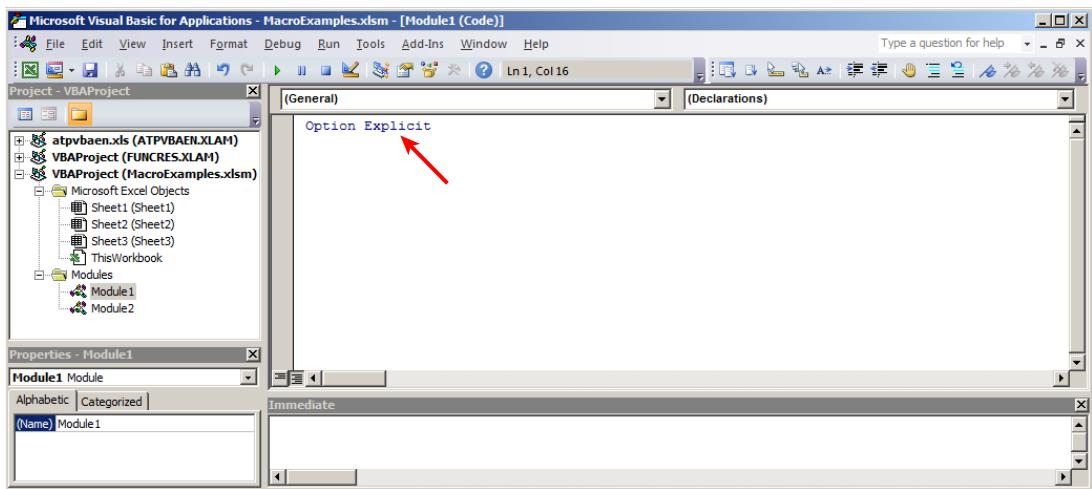


FIGURE 6-4

UNDERSTANDING A VARIABLE'S SCOPE

Variables and constants (explained in the next section) do not live forever in memory. They have a set lifetime and visibility within macros and modules. A variable's lifetime begins when it is declared, and ends when the macro that declared the variable completes its execution.

Local Macro Level Only

The visibility of a variable or constant also depends on how it is declared. If declared within a macro, a variable can only be used by that macro. For example, when Macro1 is run, the intSum variable would be calculated to a result of 41 (by adding 10 to the intAdd variable of 31), and that is what the message box would show:

```
Sub Macro1()
    Dim intAdd As Integer, intSum As Integer
    intAdd = 31
    intSum = intAdd + 10
    MsgBox intSum
End Sub
```

If you attempted to run another macro with same-looking but undeclared variables, you would receive a message box with the `Compile error` prompt, as shown in Figure 6-5 for Macro2. Just because a variable is declared in a macro elsewhere does not mean that VBA will recognize that same-looking variable in another macro.

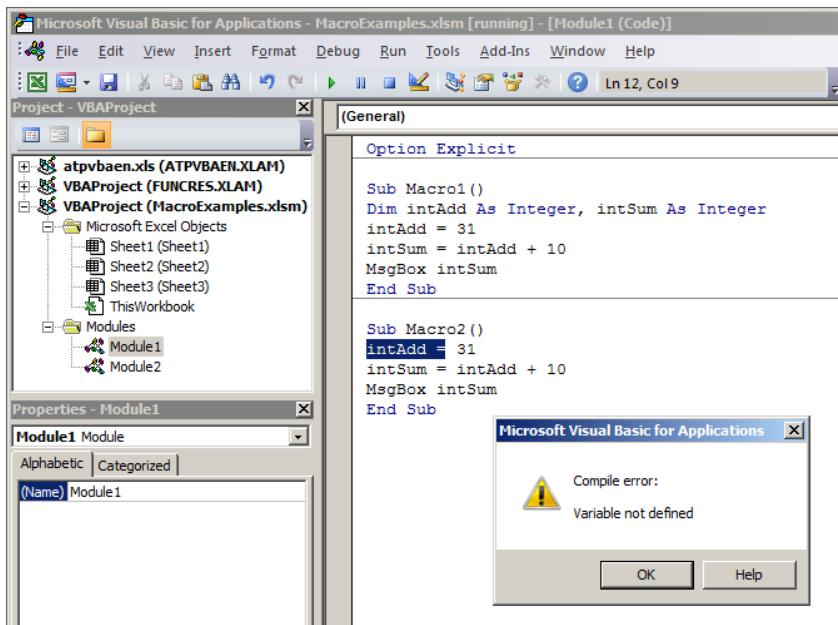


FIGURE 6-5

Module Level

It is possible for a variable to be usable in the same module by more than one macro, by having the declaration statement at the top of the module instead of inside a particular macro. In Figure 6-6, both Macro1 and Macro2 can utilize the intSum and intAdd variables.

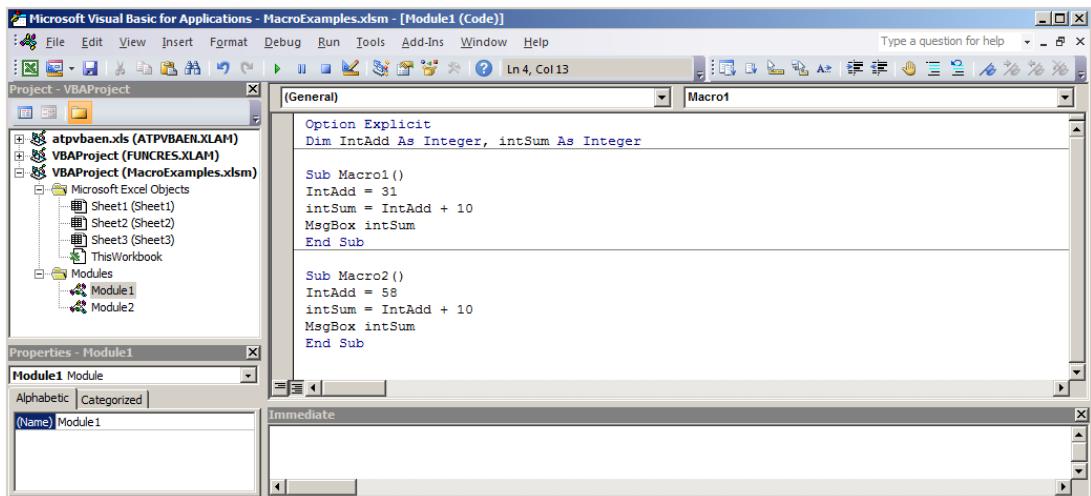


FIGURE 6-6

Application Level

Finally, you can declare the variables as `Public`, which will make them visible to all macros in all modules. You only need to place the statements at the top of one standard module, like so:

```
Public intAdd As Integer
Public intSum As Integer
```

Constants

A variable's value may often change during a macro's execution, but some macros are better served with a reference to a particular value that will not change. A *constant* is a value in your macro that does not change while the macro is running. Essentially, constants are variables that do not change.

When you declare a constant, you do so by entering a declaration statement that starts with the `Const` statement, followed by the constant's name you specify, then the data type, and finally the value, all on one line. Here is an example:

```
Const myMonths as Integer = 12
```

It's a good practice to use constants for the same reasons you would use a variable. Instead of hard-coding the same value in your macro over and over, you define the constant just once and use the reference as you need to. For example, your macro may be analyzing the company's sales amounts, and needing to factor in the sales tax at various points in the macro. This constant statement at the start of the macro would allow you to reference the 8.25% sales tax:

```
Const SalesTax as Double = .0825
```

NOTE After you declare a constant in the macro, you cannot assign a different value to it later in the macro. If you need the value to change during the macro, what you really need is a variable instead of a constant.

Choosing the Scope and Lifetime of Your Constants

The scope and lifetime of constants are much the same as for variables:

- For the constant to be available only to a particular macro, declare the constant within that macro.
- For the constant to be available only to the macros that are housed in the same module, declare the constant at the top of that module, above and outside all macros.
- For the constant to be available to all macros in all modules, prefix the constant declaration with the `Public` statement, and set it at the top of a standard module, above and outside all macros. For example:

```
Public Const SalesTax as Double = .0825
```

TRY IT

In this lesson you practice creating a macro that includes a declared variable.

Lesson Requirements

To get the sample workbook file, you can download Lesson 6 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

For this lesson you create a macro, without using the Macro Recorder, in which you declare a variable for the `String` data type, and you manipulate the string text with a few lines of practice code.

1. Create a macro that includes the following actions:
 - a. Declare a `String` type variable.
 - b. Assign text to the `String` variable.
 - c. Populate a range of cells with the `String` variable's text.
2. Open Excel and add a new workbook.
3. In your active worksheet, enter the text `Hello` in cell A1.

4. Press Alt+F11 to get into the Visual Basic Editor.
5. From the VBE menu, click Insert \Rightarrow Module.
6. In the new module, type in the name of your macro as **Sub Test6**.
7. Press the Enter key, which will cause Excel to place a set of parentheses after the **Test6** macro name, and also will create the **End Sub** statement. Your macro so far will look like this:

```
Sub Test6()
```

```
End Sub
```

8. In the empty line between **Sub Test6()** and **End Sub**, type **Dim myString As String** and press Enter.
9. Now is the time to define the **myString** variable by telling VBA that it will be equal to the value in cell A1, which is the word Hello you entered in Step 3. To do that, type the following line of code into your macro and press Enter:

```
myString = Range("A1").Value
```

10. With your **String** variable defined, try entering its defined text into a few cells, starting with cell B3. If you combine the variable with a space and the word “World,” you can programmatically enter the text “Hello World” into B3. To do that, type this line of code into your macro and press Enter:

```
Range("B3").Value = myString & " World!"
```

11. Just for fun, repeat the variable’s text three times in succession, which would be HelloHelloHello, and tell VBA to enter that into cell B4. For the next line in your macro, type **Range("B4").Value = myString & myString & myString** and press Enter.
12. As a third and final entry, show the text Hello and Goodbye in cell B5 by typing this last line of code into your macro:

```
Range("B5").Value = myString & " and Goodbye".
```

At this point, your macro is completed, and it will look like this:

```
Sub Test6()
    Dim myString As String
    myString = Range("A1").Value
    Range("B3").Value = myString & " World!"
    Range("B4").Value = myString & myString & myString
    Range("B5").Value = myString & " and Goodbye"
End Sub
```

13. Press Alt+Q to return to your worksheet.
14. Watch your new macro in action. Press Alt+F8 to display the Macro dialog box.

15. Select the Test6 macro name in the large window, as shown in Figure 6-7, and click the Run button.

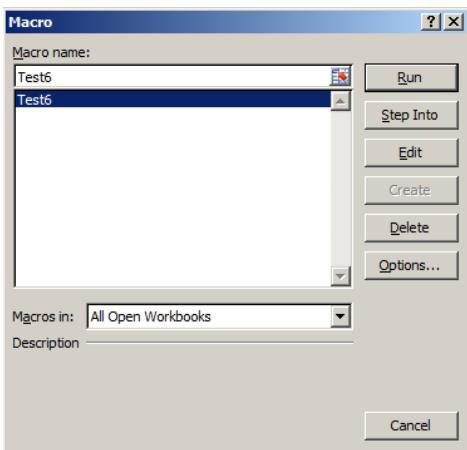


FIGURE 6-7

REFERENCE Please select the video for Lesson 6 at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

7

Understanding Objects and Collections

Lesson 5 introduced the topic of collections, which are objects that contain a group of like objects. This lesson adds some detail to the topic and goes over some programming techniques to deal with the most common types of object collections you will encounter: workbooks, worksheets, cells, and ranges.

WORKBOOKS

An Excel file is a `Workbook` object. You might wonder how workbooks have a collection, seeing as you can only work in one workbook at a time, and even then you are usually manipulating objects at a lower level, such as worksheets or cells.

NOTE *Do not confuse the Application object with the Workbook object. In VBA, the Application object is at the very top of the food chain; there is nothing higher than Application in the Excel object model. Application represents the entire Excel program, whereas Workbook represents an individual Excel file.*

The `Workbooks` collection contains the references to every `Workbook` object that is open in the same instance of Excel. You need to call upon the `Workbooks` collection when you want to do some task in every open workbook, or when you want to activate a particular workbook whose name is not known.

Here is an example. In VBA, the following command adds a new workbook:

```
Workbooks.Add
```

When this code line is executed, the active workbook becomes the new workbook you added, same as the effect of manually adding a new workbook from your existing one, when the workbook you added becomes the active workbook.

What if your project calls for you to add two workbooks to the existing one, and you want to end the macro with the first added workbook being the active one, instead of the last added workbook being the active one? In your `Workbooks` collection, how do you specify which `Workbook` object you want to do something with when you don't know the names of any open workbooks?

VBA offers several methods to solve this problem, one being an ability to assign a variable to each workbook you add, and then to activate the workbook whose variable you care about. For example, the following macro adds two workbooks and ends with the first added workbook being the active one:

```
Sub AddWorkbooks()
    Dim WorkbookAdd1 As Workbook
    Dim WorkbookAdd2 As Workbook
    Set WorkbookAdd1 = Workbooks.Add
    Set WorkbookAdd2 = Workbooks.Add
    WorkbookAdd1.Activate
End Sub
```

Workbook objects have a number of methods, such as `Open`, `Save`, and `Close`. Lesson 10 delves into the practice of repeating actions with loops, but here's a sneak peek at a loop that saves and closes every workbook that is currently open in your instance of Excel, except for the workbook you are working in. Notice what you don't see, which is a concern about how many workbooks are open, or what their names are. You only need to tell VBA to look for `Workbook` objects in the `Workbooks` collection:

```
Sub CloseAllOtherWorkbooks()
    Dim wkb As Workbook
    For Each wkb In Workbooks
        If wkb.Name <> ThisWorkbook.Name Then
            wkb.Close SaveChanges:=True
        End If
    Next wkb
End Sub
```

The `Worksheets` collection enables you to refer to the `Worksheet` objects' names or index numbers, which is the numerical position of worksheets as you see their tabs in order from left to right.

Referring to names tends to be a safer practice, but as you saw with workbooks, and as you will learn with looping techniques, a variable can be assigned to each `Worksheet` object to access all worksheets without caring where they are in the workbook or what their tab names are.

Say you want to add a new worksheet, and give it the name `Test1`. No problem there, but now you are asked to add the new worksheet such that its placement will be the last (rightmost) worksheet in the workbook. You have no idea how many sheets exist already. You don't know the name of the last worksheet in order to reference its location but even if you did know that today, there could easily be a differently named worksheet in that index position tomorrow.

The following one-line macro adds a new worksheet, names it as you specify, and places it at the far right end of the worksheets, which is the highest worksheet index number based on the count of existing worksheets:

```
Sub WorksheetTest1()
    Worksheets.Add(After:=Worksheets(Worksheets.Count)).Name = "Test1"
End Sub
```

You can place a worksheet relative to another worksheet's name, this time adding a worksheet, and placing it before Sheet2:

```
Sub WorksheetTest2()
    Worksheets.Add(Before:=Worksheets("Sheet2")).Name = "Test2"
End Sub
```

NOTE *The preceding examples work without any problem as long as the workbook does not already contain a worksheet with the name Test1 or Test2. Excel does not allow worksheets to be given duplicate names in the same workbook, and attempting to do so will result in an error. You learn about handling VBA errors in Lesson 20.*

You may want to relocate an existing worksheet from its current position to a particular index position for the convenience of your workbook's users. Suppose that during the course of your macro, you want the active worksheet to occupy the number two worksheet index position—that is, to be the worksheet that is located second from the left as you see the worksheet tabs. To accomplish this, you can place the active worksheet after the first index worksheet, as shown in the following example:

```
ActiveSheet.Move After:=Sheets(1)
```

NOTE *A word of caution about the Worksheets collection: There is a difference between the Sheets collection and the Worksheets collection. You probably know about chart sheets, and if your workbook has one, you need to be mindful to cycle through the Worksheets collection only if you are interested in manipulating worksheets. If you cycle through the Sheets collection, all sheets, including a chart sheet (or outmoded dialog sheets or macro sheets) are included in the procedure. If you only want to act on worksheets, specify the Worksheets collection.*

CELLS AND RANGES

The Range object is probably the most utilized object in VBA. A range can be a single cell or a range of cells that spans any size area. A Range object, then, is a cell or block of cells that is contained on a Worksheet object. Though a Range object can be a union of several noncontiguous blocks of cells, it is always the case that a VBA Range object is contained on a single parent worksheet. That parent worksheet can be the active worksheet or some other worksheet, but there is no such thing as a Range object that includes cells on different worksheets.

A single cell is a range as far as VBA is concerned, and ActiveCell is the object name in VBA of the single active cell on the active worksheet. There is no such object as ActiveRange, but many ways exist to identify particular ranges, one of the most common being the Selection object.

If you were to select any range of cells, and execute this line of code, all cells in that selected range would immediately contain the word “Hello”:

```
Selection.Value = "Hello"
```

You may be interested to know that named ranges are fair game for VBA to refer to and manipulate, just like any other range. In fact there is a `Names` collection object for named ranges.

As an example, say you have previously named a range `myRange`. This line of code in a VBA macro would place the word “Hello” in all cells in your named range:

```
Range("myRange").Value = "Hello"
```

As you have seen, you do not need to select your range in order to work with it. For most operations on cells or ranges, you can refer to the range and its parent worksheet. You can execute the following line of code from any worksheet in your workbook, as an example of establishing a bold format for a range of cells on Sheet1:

```
Worksheets("Sheet1").Range("A1:D25").Font.Bold = True
```

At times you will want to refer to all the cells on a worksheet instead of limiting your operation to a particular range. For example, suppose as part of your macro you want to clear the contents of every cell on the worksheet. Starting with version 2007, clearing the contents of the entire grid of worksheet cells can be expressed as `Range("A1:XFD1048576").ClearContents`. However, if the workbook is being used in a version of Excel prior to 2007, that same operation could be expressed as `Range("A1:IV65536").ClearContents`. Fortunately, you can avoid errors and confusion by using the `Cells` object as shown in the following example, which refers to all worksheet cells in whichever version of Excel is being used at the moment:

```
Cells.ClearContents
```

You can do some useful operations using the `Cells` object when you want to involve the entire worksheet. Suppose you have set up Sheet1 as a template with formatted ranges, labels, values, and formulas, and you want Sheet2 to be established the same way. The following line of code copies the Sheet1 cells and pastes them to Sheet2:

```
Worksheets("Sheet1").Cells.Copy Worksheets("Sheet2").Cells
```

SpecialCells

An interesting brand of range objects is Excel’s group of `SpecialCells`, which I touched upon in the Try It section and video for Lesson 2. Press the F5 key to show the Go To dialog box. Click the Special button, and you see more than a dozen types of `SpecialCells`.

`SpecialCells` is the name of the method in VBA that returns a range object of a specific cell type. For example, cells on your worksheet that contain comments are regarded by VBA as `SpecialCells`. So are cells containing data validation, or cells that contain formulas, or cells that contain constants, such as text or data you have manually entered. With the combinations of `SpecialCells`, the possibilities are enormous for identifying various kinds of ranges based on all sorts of criteria.

Say in range A1:A10 you have some cells that contain formulas, some cells that contain numbers you have manually entered, and some cells that contain nothing. If you want to select all individual cells in range A1:A10 that contain formulas, and not include in your selection any of the other cells in that range, this macro would do that:

```
Sub FindFormulas()
    Range("A1:A10").SpecialCells(xlCellTypeFormulas).Select
End Sub
```

TRY IT

In this lesson you practice with the useful IntelliSense tool to help you become familiar with VBA syntax. Using IntelliSense can help improve your efficiency and accuracy, with its drop-down list of properties and methods when writing code.

Lesson Requirements

None

Step-by-Step

VBA's IntelliSense feature is an incredibly useful tool that helps you write your macros faster and smarter. I use it all the time to help me write code in the proper VBA syntax. As mentioned, VBA has hundreds of objects and each object can have dozens of methods and properties. IntelliSense can display a list of an object's methods and properties while you are typing your code, and it can quickly call the Help feature for a topic you select.

1. Open Excel and press Alt+F11 to go to the Visual Basic Editor.
2. If you have not already done so, from the VBE menu bar, click Tools → Options as shown in Figure 7-1.

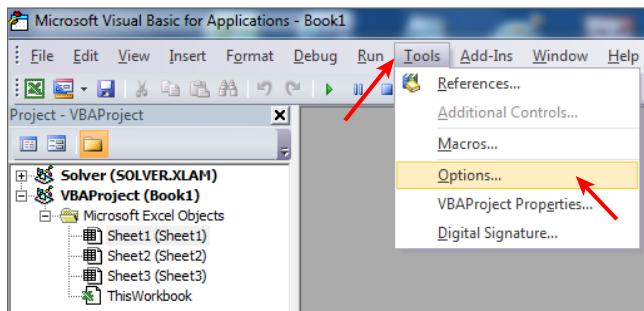


FIGURE 7-1

3. In the Options dialog box on the Editor tab, make sure there is a check mark in the box next to Auto List Members as shown in Figure 7-2, and click OK.

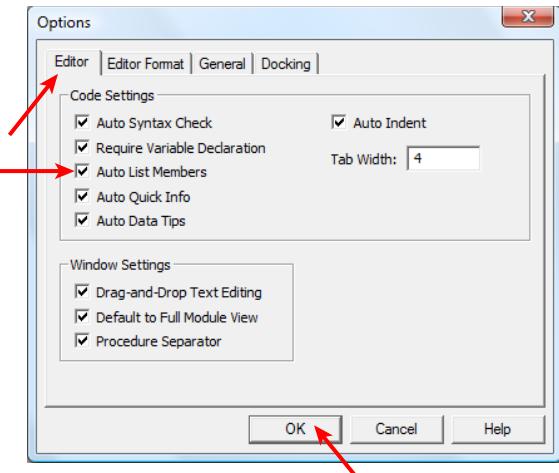


FIGURE 7-2

4. Press **Ctrl+G** to be taken into the Immediate window.
5. Type in the question mark character, then press the spacebar, type the word **Application**, and press the dot key on your keyboard. A list of the Application object's members, properties, and methods is displayed, as shown in Figure 7-3.

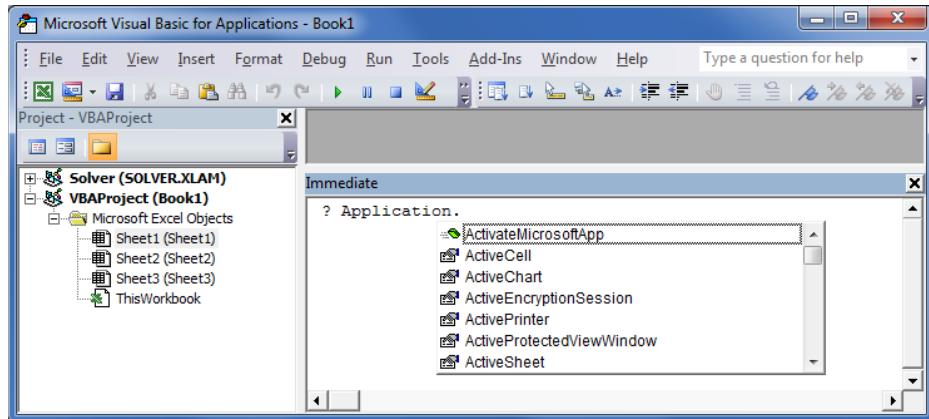


FIGURE 7-3

6. Now, practice using IntelliSense. Press the **N** key and you are taken to the first item in the Application object's list of members that begins with the letter **N**. In this case, that member happens to be the **Name** property, which will be highlighted by selection as shown in Figure 7-4.
7. With the **Name** property item selected, either double-click it or press the **Alt** key to accept and enter the **Name** property for the **Application** object, and then press the **Enter** key. The Immediate window returns the result **Microsoft Excel** as shown in Figure 7-5.

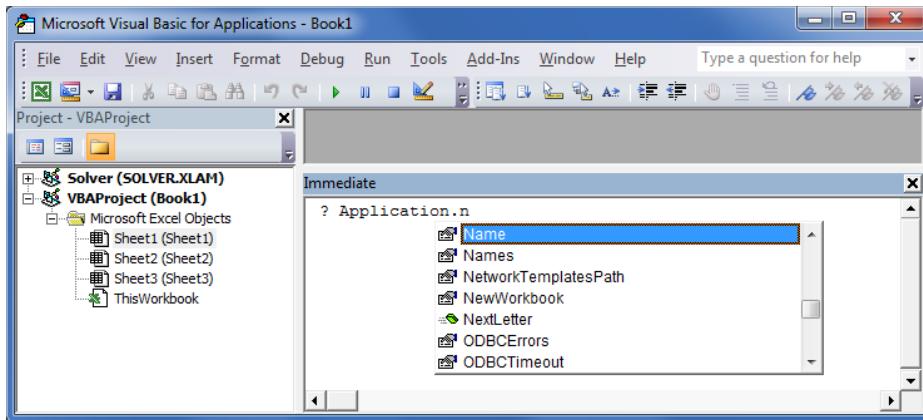


FIGURE 7-4

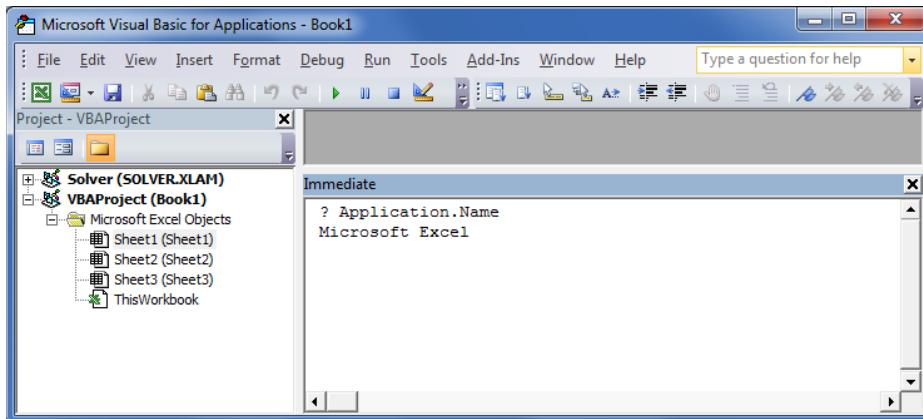


FIGURE 7-5

8. Continue to explore on your own. Press the Enter key in the Immediate window to start a new line, enter the question mark character and press the spacebar, and scroll through the member list of other objects such as ActiveWorkbook or Range. Keep in mind that many objects are parents of other objects, so you can go two or more members deep to gather some information. For example, the ActiveWorkbook object has a Worksheets collection, and the Worksheets collection has a Count property. Therefore, if you type the line ? activeworkbook.Worksheets.Count into the Immediate window, VBA returns the number of worksheets the active workbook contains.

REFERENCE Please select the videos for Lesson 7 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

8

Working with Ranges

The Range object is probably the most frequently used object in VBA. Almost anything you do in a worksheet where VBA is concerned involves either a cell (or range of cells), or a reference of some kind to a range location that helps to direct whatever action your macro is undertaking. When you programmatically create a chart, modify a pivot table's source data, or insert picture files or comments, you are working with ranges.

Because ranges are so commonly referred to in code, this lesson introduces you to various syntaxes you will soon become familiar with, and in fact depend on, to refer to or manipulate Range objects. My approach with this lesson is to demonstrate basic code lines with pictures to show how ranges can be identified or selected.

As you'll hear over and over in VBA programming circles, you need not, and normally will not, actually select a range in order to work with it. You can refer to and manipulate (such as by editing or formatting) ranges of cells on other worksheets or other workbooks without leaving your active worksheet. The pictures in this lesson show the selection of ranges for visual confirmation of the code at work, but after this lesson, you will rarely see code that selects or activates an object.

WORKING WITH CONTIGUOUSLY POPULATED RANGES

The simplest ranges to deal with are those that have all cells filled with data or formulas, and no empty cells within that range. Figure 8-1 shows a typical-looking list of data for which you can easily identify its last row, its last column, and its address. Based on Figure 8-1, the variables in the following macro yield 6 for the LastRow variable, and 3 (column C, the third column in the spreadsheet grid) for the LastColumn variable:

```
Sub Find_LastRow_LastColumn()
    Dim LastRow As Long, LastColumn As Long
    LastRow = Cells(Rows.Count, 1).End(xlUp).Row
    LastColumn = Cells(2, Columns.Count).End(xlToLeft).Column
    MsgBox _
        "The last row is: " & LastRow & vbCrLf & _
        "The last column is: " & LastColumn
End Sub
```

	A	B	C	D	E	F
1	Student name	Coursework	Year			
2	Sue Flay	Math	Freshman			
3	Brock Lee	Science	Senior			
4	Carrie Oakey	Geography	Sophomore			
5	Jerry Atrick	Math	Freshman			
6	Mike Raffone	History	Junior			
7						
8						
9						

FIGURE 8-1

Using the Cells Property

You have seen with the previous example and in other VBA expressions that the `Cells` property can select or refer to a range. The `Cells` range syntax is `Cells(rowIndex, columnIndex)`. Therefore, the expression `Cells(2, 5)` refers to cell E2 because that is the same as row 2 of column 5. For the `Cells` property, the row component must be a numeral, but the column can be a letter that must be in quotes, for example `Cells(2, "E")`. Therefore, in practice, either of these expressions bold cell E2:

```
Cells(2, 5).Font.Bold = True
Cells(2, "E").Font.Bold = True
```

You can incorporate two `Cells` properties within the `Range` statement to refer to a range larger than just one cell. Using the example variables for `LastRow` and `LastColumn`, the following line of code tells you the range address of the list shown in Figure 8-1:

```
MsgBox Range(cells(1, 1), Cells>LastRow, LastColumn).Address
```

As a final example, you can use `Cells` to select a particular range of cells, such as D3:F5 in Figure 8-2:

```
Range(cells(3, 4), cells(5, 6)).select
```

	A	B	C	D	E	F	G	H
1	Data							
2	Data						Data	Data
3	Data		Data	Data	Data	Data	Data	Data
4	Data		Data	Data	Data	Data	Data	Data
5	Data		Data	Data	Data	Data	Data	Data
6	Data						Data	Data
7	Data		Data	Data	Data	Data	Data	Data
8								

FIGURE 8-2

Using CurrentRegion

The `CurrentRegion` property refers to a localized range of contiguous data that may exist among other ranges on that worksheet containing a mix of filled and empty cells. Suppose you have disjointed data on your worksheet, as depicted in Figure 8-3. With the active cell in the local

(CurrentRegion) area of the range of data you want to work with, the following line of code will select that active cell's CurrentRegion, as shown in Figure 8-4.

```
ActiveCell.CurrentRegion.Select
```

	A	B	C	D	E	F	G	H
1			Data	Data	Data	Data	Data	
2	Data						Data	
3	Data		Data	Data	Data		Data	
4	Data		Data	Data	Data		Data	
5	Data		Data	Data	Data		Data	
6							Data	
7			Data		Data		Data	
8								

FIGURE 8-3

	A	B	C	D	E	F	G	H
1			Data	Data	Data	Data	Data	
2	Data						Data	
3	Data		Data	Data	Data		Data	
4	Data		Data	Data	Data		Data	
5	Data		Data	Data	Data		Data	
6							Data	
7			Data		Data		Data	
8								

FIGURE 8-4

WORKING WITH NONCONTIGUOUSLY POPULATED RANGES

You will often need to locate or refer to ranges that are broken up by empty cells, usually referred to as *noncontiguous ranges*. VBA offers some clever options for taming the noncontiguous range beast.

Using Range with Several Cells

As shown in Figure 8-5, you can select various cells in a union with this example code line:

```
Range("B2, D5, F1:F4").Select
```

Notice the construction has cell addresses, or ranges, separated by a comma and a space, enclosed in quotes.

	A	B	C	D	E	F	G	H
1			Data	Data	Data	Data	Data	
2	Data						Data	
3	Data		Data	Data	Data		Data	
4	Data		Data	Data	Data		Data	
5	Data		Data	Data	Data		Data	
6							Data	
7			Data		Data		Data	
8								

FIGURE 8-5

Using OFFSET

The `OFFSET` property refers to a range by adding or subtracting (offsetting) row and column numbers from a relative reference to refer to a new range. In Figure 8-6, the active cell is B4.

	A	B	C	D	E	F	G	H
1			Data	Data	Data	Data	Data	
2	Data						Data	
3	Data		Data	Data	Data		Data	
4	Data		Data	Data	Data		Data	
5	Data		Data	Data	Data		Data	
6							Data	
7			Data		Data		Data	
8								

FIGURE 8-6

If you want to select a range that is relative to the active cell by extending the range upward 2 rows and outward 4 columns, you can use the following code line. The result is shown in Figure 8-7.

```
Range(activecell, activecell.Offset(-2, 4)).Select
```

	A	B	C	D	E	F	G	H
1			Data	Data	Data	Data	Data	
2	Data						Data	
3	Data		Data	Data	Data		Data	
4	Data		Data	Data	Data		Data	
5	Data		Data	Data	Data		Data	
6							Data	
7			Data		Data		Data	
8								

FIGURE 8-7

Using RESIZE

The `RESIZE` property changes the size of a range, based on a cell of interest as the reference point. In this example, range B3 is resized by 4 rows and 5 columns, thereby selecting range B3:F6. (See Figure 8-8.) The code line that is used in this example is

```
Range("B3").Resize(4, 5).Select
```

	A	B	C	D	E	F	G	H
1							Data	
2	Data		Data	Data	Data		Data	
3	Data		Data	Data	Data		Data	
4	Data		Data	Data	Data		Data	
5	Data		Data	Data	Data		Data	
6	Data		Data	Data	Data		Data	
7	Data		Data	Data	Data		Data	
8	Data		Data	Data	Data		Data	

FIGURE 8-8

Identifying a Data Range

In some cases you will only want to identify a range of cells that contain data or formulas, but not formatting. In Figure 8-9, cell H3 is a lonely soul, apart from the data range but formatted with red fill color for demonstration purposes.

This example shows how to select a data range on the current sheet, starting at cell A1, and display the address of the range to the user. The data range does not include cells that are formatted that do not contain data. To get the data range, this example finds the last row and the last column that contain actual data by using the `Find` method of the `Range` object:

```
Sub SelectDataRange()
    Dim LastRow As Long, LastColumn As Long
    LastRow = Cells.Find(What:="*", SearchDirection:=xlPrevious, _
        SearchOrder:=xlByRows).Row
    LastColumn = Cells.Find(What:="*", SearchDirection:=xlPrevious, _
        SearchOrder:=xlByColumns).Column
    Range("A1").Resize(LastRow, LastColumn).Select
    MsgBox "The data range address is " & Selection.Address(0, 0) & ".", _
        vbInformation, "Data-containing range address:"
End Sub
```

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O	P
1	Data	Data	Data													
2	Data	Data	Data	Data	Data											
3	Data	Data	Data	Data												
4	Data	Data	Data													
5	Data															
6	Data			Data												
7			Data	Data												
8		Data			Data	Data										
9																

FIGURE 8-9

Identifying the UsedRange

The `UsedRange` property represents cells on a worksheet that are currently being used or have been used. This includes formatted cells that do not contain data, such as what's shown in Figure 8-10.

This example shows how to select the `UsedRange` on the current worksheet by using the `UsedRange` property of the `Worksheet` object and the `Select` method of the `Range` object. The selected address of the worksheet's `UsedRange` is displayed in a message box:

```
Sub SelectUsedRange()
    ActiveSheet.UsedRange.Select
    MsgBox "The used range address is " & _
        ActiveSheet.UsedRange.Address(0, 0) & ".", 64, "Used range address:"
End Sub
```

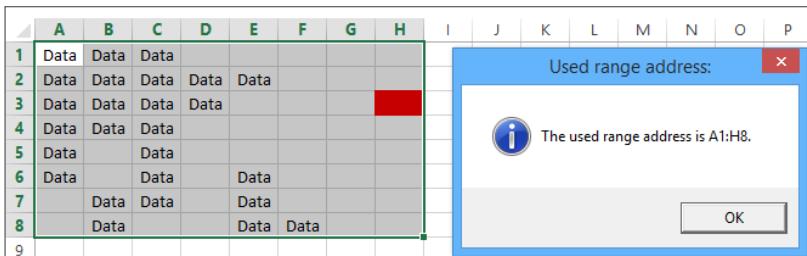


FIGURE 8-10

Finding the Dynamic Last Rows and Columns

This section includes a collection of several dynamic row and column locations wrapped into one macro example. You may need to not only find the last row of data, but limit your search to a particular set of columns. The same goes for the last used column, based on one row, all rows, or a specific range of rows. Figure 8-11 shows the versatility of the following macro for handling all these scenarios:

```
Sub DataRangeLastRowsColumns()

    'Declare variables for last rows and columns
    Dim LastRow As Long, LastColumn As Long
    Dim LastRowSingleColumn As Long, LastRowSomeColumns As Long
    Dim LastColumnSingleRow As Long, LastColumnSomeRows As Long

    'Last row of data considering all columns.
    LastRow = Cells.Find(What:="*", After:=Range("A1"), _
        SearchOrder:=xlByRows, SearchDirection:=xlPrevious).Row
    'Last row of data considering just column D.
    LastRowSingleColumn = Cells(Rows.Count, 4).End(xlUp).Row
    'Last row of data considering just columns B, C, and D.
    LastRowSomeColumns = Range("B:D").Find(What:="*", After:=Range("B1"), _
        SearchOrder:=xlByRows, SearchDirection:=xlPrevious).Row

    'Last column of data considering all rows.
    LastColumn = Cells.Find(What:="*", After:=Range("A1"), _
        SearchOrder:=xlByColumns, SearchDirection:=xlPrevious).Column
    'Last column of data considering just row 3.
    LastColumnSingleRow = Cells(3, Cells.Columns.Count).End(xlToLeft).Column
    'Last column of data considering just rows 1, 2, and 3.
    LastColumnSomeRows = Rows("1:3").Find(What:="*", _
        After:=Cells(1, Cells.Columns.Count), _
        SearchOrder:=xlByColumns, SearchDirection:=xlPrevious).Column

    'Advise the user of last row and column information.
    MsgBox _
        "Last row of data anywhere: " & LastRow & vbCrLf & _
        "Last row of data in column D: " & LastRowSingleColumn & vbCrLf & _
        "Last row of data among columns B, C, and D: " & _

```

```

    LastRowSomeColumns & vbCrLf & vbCrLf &
    "Last column of data anywhere: " & LastColumn & vbCrLf &
    "Last column of data in row 3: " & LastColumnSingleRow & vbCrLf &
    "Last column of data among rows 1, 2, and 3: " & LastColumnSomeRows, , ,
    "Last row and last column information:"
```

End Sub

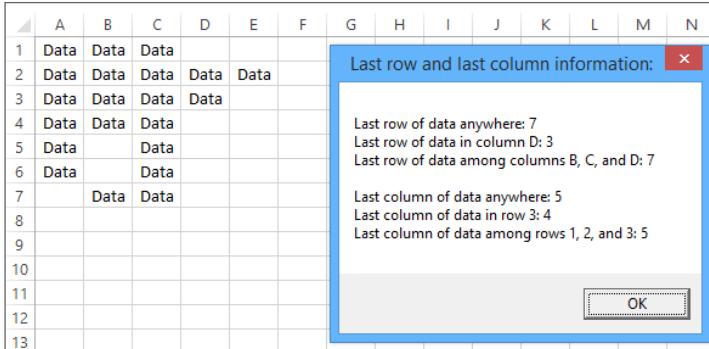


FIGURE 8-11

Identifying Where the Range Starts and Ends When No Start or End Point Is Known

It probably seems that cell A1 is where data starts on a worksheet. Row 1 is popular for header labels in a list, and column A is the leftmost column on the spreadsheet grid, prominently visible. But sometimes data finds itself on a worksheet in areas you would not expect, and the next day, that same worksheet can hold data somewhere totally different. You need a catch-all macro to find the range of data, from wherever it starts to wherever it ends.

This example shows how to select a data range on the current sheet when you do not know the starting or ending location and display the range address in a message box. The data range does not include cells that are formatted. This example finds the first and last row and column that contain actual data by using the `Find` method of the `Range` object. The result is shown in Figure 8-12.

```

Sub UnknownRange()
Dim FirstRow As Long, FirstCol As Long, LastRow As Long, LastCol As Long
Dim myUsedRange As Range

FirstRow = _
Cells.Find(What:="*", SearchDirection:=xlNext, SearchOrder:=xlByRows).Row
FirstCol = _
Cells.Find(What:="*", SearchDirection:=xlNext, SearchOrder:=xlByColumns).Column
LastRow = _
Cells.Find(What:="*", SearchDirection:=xlPrevious, SearchOrder:=xlByRows).Row
LastCol = _
```

```

Cells.Find(What:="*", SearchDirection:=xlPrevious, SearchOrder:=xlByColumns).Column

Set myUsedRange = Range(Cells(FirstRow, FirstCol), Cells(LastRow, LastCol))
myUsedRange.Select
MsgBox _
"The data range on this worksheet is " &_
myUsedRange.Address(0, 0) & ".", vbInformation, "Range address:"
End Sub

```

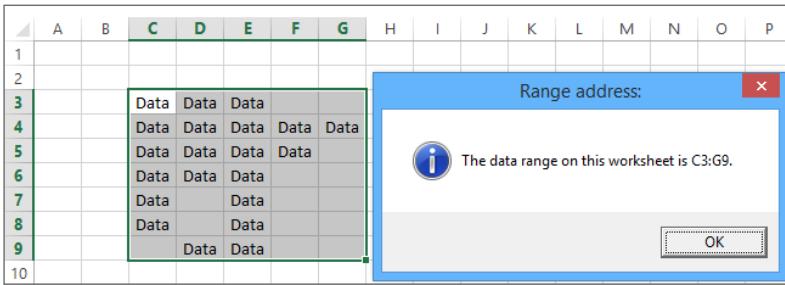


FIGURE 8-12

TRY IT

In this lesson, you see how to create a macro that identifies the location of a chart on a worksheet. The purpose of the exercise is to demonstrate how to identify an object's location without selecting or activating any ranges or objects. The value of the exercise is to know with confidence where else on a worksheet (that is, below or to the right of an object) you can insert a new object, edit a cell, or take some action on the worksheet without coming into contact with the existing object of interest.

Lesson Requirements

To get the sample workbook file, you can download Lesson 8 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

Using the `Index` property of the embedded chart helps avoid needing to know the `Chart` object's name.

The `Cells` property is especially useful in this example, when variables are declared to identify row and column locations.

Step-by-Step

1. Open an Excel workbook in which you have a `Chart` object on a worksheet. If you don't have such a workbook, in a new worksheet, construct a simple table and insert a chart similar to Figure 8-13.

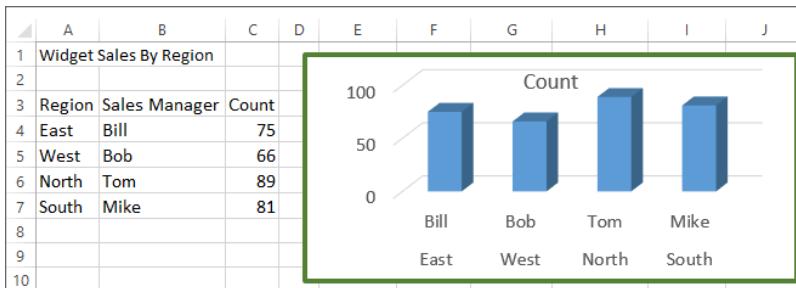


FIGURE 8-13

2. Compose a macro to tell you the location of the chart on your worksheet. You will want to identify the top and bottom rows, and the left and right columns that the Chart object covers.
3. From the Developer tab on the Ribbon, click the Visual Basic icon, or press Alt+F11 on your keyboard to go to the Visual Basic Editor.
4. From the VBE menu bar, click Insert \leftrightarrow Module.
5. In your new module, type the name of the macro. In this example, the macro is named Sub ChartLocation().
6. Declare variables for the top and bottom rows, and left and right columns that the chart touches:

```
Dim TopRow As Long, BottomRow As Long
Dim LeftColumn As Long, RightColumn As Long
```

7. Open a With structure for the ChartObject. Because there is only one chart on the worksheet, its Index property is 1 and you can refer to it in code with this statement:

```
With ActiveSheet.ChartObjects(1)
```

8. Declare your row and column variables like so:

```
TopRow = .TopLeftCell.Row
BottomRow = .BottomRightCell.Row
LeftColumn = .TopLeftCell.Column
RightColumn = .BottomRightCell.Column
```

9. Close the With structure:

```
End With
```

10. Utilizing the variables to show an example of changing a cell outside the range occupied by the chart, this line of code enters the word Hello into a cell two rows below and two columns to the right of the bottom-right corner of the chart:

```
Cells(BottomRow + 2, RightColumn + 2).Value = "Hello"
```

11. For demonstration purposes, an optional enhancement to this macro is the following message box code that confirms the chart's location when the macro is run, as shown in Figure 8-14:

```
MsgBox "Top row: " & TopRow & vbCrLf & _
"Bottom row: " & BottomRow & vbCrLf & _
"Left column: " & LeftColumn & vbCrLf & _
"RightColumn: " & RightColumn, , "ChartLocation"
```

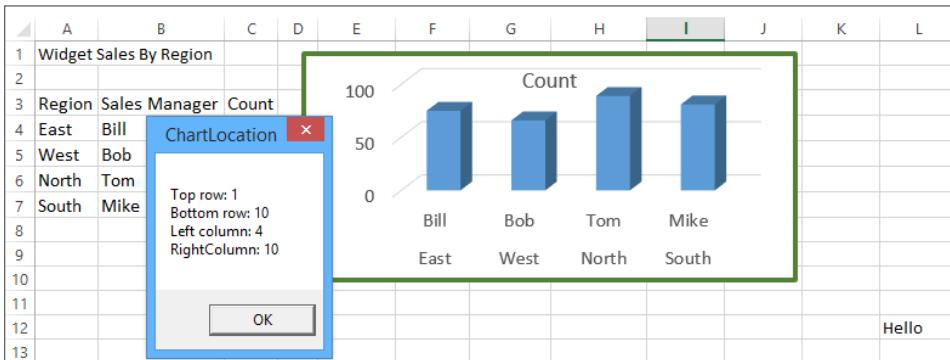


FIGURE 8-14

12. End the macro with the `End Sub` line.
 13. Go ahead and test your macro. Press Alt+Q to exit the VBE, and from your worksheet press Alt+F8 to show the Macro dialog box. Select the macro name and click the Run button. The `ChartLocation` macro looks like this in its entirety:

```
Sub ChartLocation()
Dim TopRow As Long, BottomRow As Long
Dim LeftColumn As Long, RightColumn As Long
With ActiveSheet.ChartObjects(1)
TopRow = .TopLeftCell.Row
BottomRow = .BottomRightCell.Row
LeftColumn = .TopLeftCell.Column
RightColumn = .BottomRightCell.Column
End With
Cells(BottomRow + 2, RightColumn + 2).Value = "Hello"
MsgBox "Top row: " & TopRow & vbCrLf & _
"Bottom row: " & BottomRow & vbCrLf & _
"Left column: " & LeftColumn & vbCrLf & _
"RightColumn: " & RightColumn, , "ChartLocation"
End Sub
```

REFERENCE Please select the video for Lesson 8 at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

9

Making Decisions with VBA

So far, all the macros you've created share a common trait of being executed line by line, starting with the very first line of code below the `Sub` name, and ending at the `End Sub` line. You might think that this is the very purpose of a VBA macro, for all its code lines to be run in sequence from start to finish. After all, isn't that why VBA code is in a macro in the first place?

It turns out that VBA can do a lot more with your macros than just serve the purpose of executing every line of code in them. You will encounter many instances when you'll need to guide the user into making a decision about whether to do one thing or another. There are also times when you will want VBA to just go ahead and make a decision about something, without any input from the user.

Depending on the decisions that get made during the course of a macro, you'll want VBA to execute only the code relating to the selected choice, while bypassing the alternative code relating to which choice was not selected. This lesson shows you how to ask the user for information when the situation calls for it, and also how to simply let VBA do the decision-making on the fly, in circumstances when the user does not even need to be involved in the decision process.

UNDERSTANDING LOGICAL OPERATORS

Logical operators are terms in VBA that you can use for evaluating or comparing a combination of individual expressions in order to make a decision in your macro, and for VBA to carry out the code relating to that decision. The three most commonly used logical operators are `AND`, `OR`, and `NOT`, and all three have the same logical effect in VBA as they do in Excel's worksheet functions.

To understand how and why to use these logical operators in your macro, it's important to take a look at the conditions under which each one will yield a positive (True) result or a negative (False) result. A truth table is a good way to illustrate each logical operator's True or False outcome, depending on the combinations of all possible results from the VBA expressions being compared. After you understand the theory of logical operators, you will see how to put them to practical use when your macros call for decisions to be made.

AND

The AND logical operator performs a conjunction by comparing two expressions. The result of the AND operation is True only if both conditions are True. If either or both conditions are False, the And operation evaluates to False.

For example, say you enter the number 500 in cell A1, and you enter the number 850 in cell B1. The following statement with the AND operator evaluates to True because both conditions are true at the same time:

```
Range("A1").Value > 300 AND Range("B1").Value > 700
```

Keeping the same numbers in cells A1:B1, the following statement would evaluate to False because, even though the first condition is True, the second condition is False:

```
Range("A1").Value > 300 AND Range("B1").Value > 900
```

This next statement would also evaluate to False, because even though the second condition is True, the first condition is False:

```
Range("A1").Value > 620 AND Range("B1").Value > 700
```

The final possibility is if both conditions are False, with this statement for example, which would evaluate to False:

```
Range("A1").Value < 200 AND Range("B1").Value < 700
```

Table 9-1 summarizes each possible result of the AND logical operator more succinctly.

TABLE 9-1: Truth Table for the AND Logical Operator

EXPRESSION 1	EXPRESSION 2	LOGICAL RESULT
True	True	True
True	False	False
False	True	False
False	False	False

OR

The OR operator performs a logical disjunction, whereby if either condition is True, or if both conditions are True, the result is True. If both conditions are False, the OR operation results in False. For example, using the same cell values as the previous AND example, with 500 in cell A1 and 850 in cell B1, you can see how differently the four statements will evaluate, using OR instead of AND as the logical operator.

The first statement evaluates to True, not necessarily because both conditions are True, but because at least one condition is True:

```
Range("A1").Value > 300 OR Range("B1").Value > 700
```

The following statement would evaluate to True on the strength of the first condition being True, even though the second condition is False:

```
Range("A1").Value > 300 OR Range("B1").Value > 900
```

This next statement would also evaluate to True because, despite the first condition being False, the second condition is True:

```
Range("A1").Value > 620 OR Range("B1").Value > 700
```

The final possibility is if both conditions are False, meaning that in this case, because neither condition is True, the statement would evaluate to False:

```
Range("A1").Value < 200 OR Range("B1").Value < 700
```

Table 9-2 summarizes each possible result of the OR logical operator.

TABLE 9-2: Truth Table for the OR Logical Operator

EXPRESSION 1	EXPRESSION 2	LOGICAL RESULT
True	True	True
True	False	True
False	True	True
False	False	False

NOTE Careful! Comparing logical expressions does not mean you can compare the impossible. Consider the following example:

```
Dim intNumber As Integer
intNumber = 0
MsgBox intNumber <= 5 Or 10 / intNumber > 5
```

Because it is impossible to divide a number by zero, this code produces an error even though the first condition evaluated to True.

NOT

The NOT operator performs logical negation. Similar to the negative sign in front of a worksheet formula, the NOT operator inverts an expression's True or False evaluation. For example, the following line of code toggles as on or off the display of gridlines on the active worksheet:

```
ActiveWindow.DisplayGridlines = Not ActiveWindow.DisplayGridlines
```

The logic behind this use of the NOT operator is to make the status of an object's property be opposite of whatever its current status is. In this case, the DisplayGridlines property of the ActiveWindow object can only be True (show the gridlines) or False (do not show the gridlines). Therefore, using the NOT operator in this way, you get the effect of toggling between showing and not showing the active worksheet's gridlines at each re-execution of this line of code.

Table 9-3 summarizes each possible result of the NOT logical operator.

TABLE 9-3: Truth Table for the NOT Logical Operator

EXPRESSION	LOGICAL RESULT
True	False
False	True

CHOOSING BETWEEN THIS OR THAT

This lesson began by mentioning that some code in your macros will need to be purposely bypassed. Most computer programming languages, VBA included, provide for the flexibility of structuring your code so that every command does not need to be run in every case. Many times, you will write macros wherein you will want the program to run certain commands if the user clicks Yes and alternative commands if the user clicks No. All of the commands are a part of the macro code, but only one set of them will execute.

If...Then

Among VBA's arsenal of decision-making commands, the If...Then statement is probably the simplest and most commonly utilized approach to structure your conditional scenarios. Consider this line of code:

```
If Weekday(VBA.Date) = 6 Then MsgBox "Have a nice weekend!", , "Today is Friday!"
```

If you have worked with Excel's WEEKDAY worksheet function, you may recall to Excel, weekday number 1 is Sunday, weekday number 2 is Monday, and so on. VBA would look at this line of code and display the message box only if the line of code is being executed on a Friday because Friday is weekday number 6. If the weekday is any day other than Friday, VBA bypasses this line of code.

NOTE *In your prior VBA travels, you might have only seen an If statement with an accompanying End If statement below it, and you might be wondering why and how the previous example can be successfully executed without having or needing an End If statement. The previous example could have been written in “block” style like this:*

```
If Weekday(VBA.Date) = 6 Then
    MsgBox "Have a nice weekend!", , "Today is Friday!"
End If
```

When evaluating for a single condition, and the conditional code is one task as shown in this example, you can write the entire If...Then statement in a single line of code. Some programmers prefer a single If line for their one-condition evaluations, and other programmers prefer the block style. It comes down to a personal preference and whatever feels more intuitive to you.

If...Then...Else

More often than not, your evaluations will involve two or more conditions instead of just one. When you have two conditions and each has its own set of tasks to carry out, you need to separate the two conditions with the `Else` statement in a block `If` structure.

Expanding on the previous example, say you want to display a message box if today is Friday, but a different message box if today is not Friday. Here is the format you would use in your macro:

```
If Weekday(VBA.Date) = 6 Then
    MsgBox "Have a nice weekend!", , "Today is Friday!"
Else
    MsgBox "Alas, today is not Friday.", , "Not Friday yet!"
End If
```

Notice that the `Else` statement stands alone on its own dedicated line, separating the two conditions' respective commands. Only one condition can possibly evaluate to True in this example, because today is either Friday or it is some day other than Friday. This block of code is designed to always be executed such that only one of the message box commands would appear, but never both during the same run.

NOTE *Here's a design tip to speed up your programs. In a block `If` structure with multiple conditions, VBA looks at each condition in turn, and basically stops at and executes the conditional code for the first condition that is found to evaluate to True. With two or three conditions, it might not be a big deal in which order you set your conditions in the `If` structure. But sometimes you will be programming for multiple conditions, and the point is, you will want VBA to execute its process as efficiently as possible. A good habit to get into is to design your `If` structures by setting the first condition to be the one that's most likely to be the case. That way, most of the time, the first condition will be the True condition and VBA will not waste time evaluating the alternative unlikelier scenarios. With this in mind, the previous example is a good opportunity to show how to make your code run faster. You can see that the first condition dealt with the current weekday being Friday. If you think about it, there is only one chance in seven that will be the case. Mostly, the macro will be run on one of the other days of the week. A better way to write the `If` code is to consider which condition will be True more often than the other condition(s). Six out of seven days will not be a Friday, so that condition should be placed first, as shown in this example:*

```
If Weekday(VBA.Date) <> 6 Then
    MsgBox "Alas, today is not Friday.", , "Not Friday yet!"
Else
    MsgBox "Have a nice weekend!", , "Today is Friday!"
End If
```

If...Then...ElseIf

VBA provides an extended way to utilize the `If...Then...Else` conditional structure when more than two conditions must be evaluated. Say you want to display a custom message for every day of the traditional five-day work week. You need a way to express your conditions in a single `If` structure with five possible courses of action, depending on which day of the week the macro is run.

One way you can accomplish this is with an `If...Then...ElseIf` structure as shown in the following example. Recall from the discussion about logical operators at the beginning of this lesson that you can evaluate two or more conditions in one line of code. Notice that the first five conditions coincide with the five workdays from Monday to Friday. The final condition uses the `Or` operator to identify a weekend day of either Saturday or Sunday:

```
Sub WeekdayTest()
    'Monday
    If Weekday(VBA.Date) = 2 Then
        MsgBox "Ugghhh -- Back to work.", , "Today is Monday"
    'Tuesday
    ElseIf Weekday(VBA.Date) = 3 Then
        MsgBox "At least it's not Monday anymore!", , "Today is Tuesday"
    'Wednesday
    ElseIf Weekday(VBA.Date) = 4 Then
        MsgBox "Hey, we're halfway through the work week!", , "Today is Wednesday"
    'Thursday
    ElseIf Weekday(VBA.Date) = 5 Then
        MsgBox "Looking forward to the weekend.", , "Today is Thursday"
    'Friday
    ElseIf Weekday(VBA.Date) = 6 Then
        MsgBox "Have a nice weekend!", , "Today is Friday!"
    'Saturday or Sunday
    ElseIf Weekday(VBA.Date) = 7 Or Weekday(VBA.Date) = 1 Then
        MsgBox "Hey, it's currently the weekend!", , "Today is a weekend day!"
    End If
End Sub
```

IIF

Yes, you read that correctly, that's an `IF` with an extra I, spelled `IIF`. Though it is similar in syntax to the familiar `IF` worksheet function, `IIF` is a lesser known and lesser utilized conditional function in VBA.

Why did Microsoft develop the `IIF` function? For the same reason elite swimming champions have swum the English Channel—because they could. You'll see `IIF` being used about as many times as the English Channel has been swum, which is not many, but I am including it here so you can say you know about it if the subject should come up at the water cooler.

The syntax for the `IIF` construction is `IIF(Expression, TruePart, FalsePart)`.

Recall from earlier in this lesson that my example for `If...Then...Else` was this five-line construction:

```
If Weekday(VBA.Date) = 6 Then
    MsgBox "Have a nice weekend!", , "Today is Friday!"
Else
```

```
MsgBox "Alas, today is not Friday.", , "Not Friday yet!"
End If
```

The `IIf` function can handle all that in one line, like this:

```
MsgBox IIf(Weekday(VBA.Date) = 6, "Today is Friday!", "Not Friday yet!")
```

Getting used to `IIf`'s syntax and appearance is an acquired taste that most VBA programmers don't pursue. Beyond that, however, is the risky and inefficient nature of `IIf` whereby both the `TruePart` and `FalsePart` statements are evaluated, even if the `TruePart` evaluates to True. Also, if your `FalsePart` statement should evaluate to an error, such as dividing a number by zero or referring to a named range that does not exist, the entire `IIf` statement will result in an error, even if the `TruePart` statement is True.

Select Case

As you are fully aware, the world is a complicated place and your macros will sometimes need to take into consideration not just one, two, or five courses of action, but possibly ten, hundreds, or even thousands depending on the situation. There are also times when several possible different conditions will require the same course of action. For these complex evaluations, the `Select Case` statement is a perfect solution.

You will want to become familiar with `Select Case`. It is simple to use, and it is easier to follow in your code than an extensive `If` structure. Similar to the `If` and `ElseIf` keywords, you use the `Case` keyword in a `Select Case` structure to test for the True evaluation of a particular condition or set of conditions. You can have as many `Case` statements as you want, and only the code associated with the first `Case` that evaluates to True will be executed.

The best way to understand `Select Case` is to see it in action with a few examples. The following macro named `WeekdayTestSelectCase` is actually the previous `WeekdayTest` macro, which accomplishes the same result, but uses `Select Case` structure instead of `If...Then...ElseIf`:

```
Sub WeekdayTestSelectCase()
Select Case Weekday(VBA.Date)
Case 2 'Monday
MsgBox "Ugghhh - - Back to work.", , "Today is Monday"
Case 3 'Tuesday
MsgBox "At least it's not Monday anymore!", , "Today is Tuesday"
Case 4 'Wednesday
MsgBox "Hey, we're halfway through the work week!", , "Today is Wednesday"
Case 5 'Thursday
MsgBox "Looking forward to the weekend.", , "Today is Thursday"
Case 6 'Friday
MsgBox "Have a nice weekend!", , "Today is Friday!"
Case 1, 7 'Saturday or Sunday
MsgBox "Hey, it's currently the weekend!", , "Today is a weekend day!"
End Select
End Sub
```

You'll notice less redundancy of each condition (each `Case`), because the primary item of interest, `Weekday(VBA.Date)`, needs to be named only once in the `Select Case` statement, instead of in every `ElseIf` statement. Also, each `Case` is very clear, and the entire macro is just easier to read.

A useful tactic with `Select Case` is the ability to group several different conditions into a single `Case` if it satisfies a particular test. For example, if your company operates its budget on a calendar-year basis, that means the months of January, February, and March belong to Quarter 1; April, May, and June belong to Quarter 2, and so on.

With the `Select Case` structure, you can group different conditions into the same `Case` to arrive at a common result. It is not just that January has a one-to-one association with Quarter 1, because the months of February and March also comprise Quarter 1. If you want to produce a message box that displays the current quarter, this macro shows how to group the months into cases:

```
Sub CurrentQuarter()
Select Case Month(VBA.Date)
Case 1 To 3: MsgBox "Quarter 1"
Case 4 To 6: MsgBox "Quarter 2"
Case 7 To 9: MsgBox "Quarter 3"
Case 10 To 12: MsgBox "Quarter 4"
End Select
End Sub
```

As you can see, you don't need 12 separate statements to handle each conditional month; you can simply state the range of months using the `To` statement in each `Case`. I put a new wrinkle in that macro to point out a VBA feature, that being the colon character (`:`), which can be used to separate multiple statements on the same `Case` line that would otherwise each require their own line. I don't usually use the colon character this way, but sometimes it comes in handy by helping the readability of small macros like this.

Here's a final example while we're on this topic, to show how useful the `Select Case` structure is when the cases can include thousands of items that can all satisfy a `Case` criteria. Suppose the management of a football squad wants to enter the paid attendance of today's game into cell A1, and run a macro to assess the fans' paid attendance. You can see how valuable `Select Case` can be, if, say 85,000 people attended the game, with that situation being handled with mathematical operators in your `Case` statements:

```
Sub SelectCaseExample()
Dim PaidAttendance As Long
PaidAttendance = Range("A1").Value
Select Case PaidAttendance
Case Is < 1000: MsgBox "Small-sized crowd!"
Case Is < 5000: MsgBox "Medium-sized crowd!"
Case Is >= 5000: MsgBox "WOW! Excellent! Huge crowd!"
End Select
End Sub
```

GETTING USERS TO MAKE DECISIONS

Thus far you have seen examples of VBA's decision-making abilities that have not required any input from the user. The time will come when you'll either want or need information from the user in order for decisions to be made that only the user can provide. Message boxes and input boxes are excellent tools to interact with your users in such situations.

Message Boxes

Up to this point in the book, you have seen many examples of code that include a message box. In all those examples, the message box was a simple pop-up box that displayed an informational text message, with an OK button for you to acknowledge the information.

Message boxes are flexible tools that allow you to customize the buttons while asking questions directly to the users that will force them to select one option or the other. Instead of OK, you can display a Yes button and a No button on your message box, and write the code that will be followed if the user clicks Yes or the user clicks No. An example of such a message box is shown in Figure 9-1.

Say you have a macro to perform a task that your users should confirm they really want to do as a final OK. Some macros are quite large and virtually irreversible, or the task at hand will alter the workbook in a significant way. In the following simplified example, the active worksheet will be copied and placed before Sheet1, but only if the user first clicks the Yes button to confirm his intention for this to happen. If the user clicks No, a friendly message box advises the user that the macro will not run because No was clicked:

```
Sub ConfirmExample()
Select Case MsgBox(
    "Do you really want to copy this worksheet?", _
    vbYesNo + vbQuestion, _
    "Please confirm...")

Case vbNo
MsgBox _
    "No problem, this worksheet will not be copied.", _
    vbInformation, _
    "You clicked No."
Exit Sub
Case vbYes
MsgBox _
    "Great -- click OK to run the macro.", _
    vbInformation, _
    "Thanks for confirming."
ActiveSheet.Copy Before:=Sheets("Sheet1")
End Select
End Sub
```

As you look at the `MsgBox` line, note that the message box arguments are contained within parentheses. A message box has two mandatory arguments: the prompt, which is the text you place in the body of the message box, and the button configuration. Other combinations of buttons include `OKCancel`, `YesNoCancel`, and `AbortRetryIgnore`. The title of the message box is optional, but I always enter it to offer a more customized experience for the user.

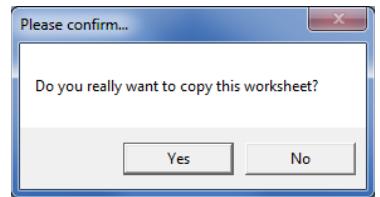


FIGURE 9-1

NOTE In the Try It section at the end of Lesson 7, you worked with VBA's IntelliSense feature. I recommend you activate IntelliSense if you have not already done so, because when composing message boxes, you'll be reminded of the available arguments and their proper syntax while you are writing your code.

Input Boxes

When you need a piece of specific information from the user, such as a text string or a number, an `InputBox` was made for the job. An *input box* looks like a distant cousin of a message box, with the prompted text that tells the user what to do, OK and Cancel buttons (which cannot be reconfigured as a message box's buttons can), and an optional title argument.

An `InputBox` requires a prompt argument, and it provides a field wherein the user would enter the kind of information as needed for the macro to continue. The entry would return a `String` type variable. If no entry is made, that is, the text field is left empty, the `InputBox` would return a null string, which is usually regarded by VBA the same as if the user clicked the Cancel button.

The following example uses an input box to ask the user to enter a number to represent how many rows will be inserted below the active cell's row. Figure 9-2 shows what the input box looks like for this macro.

```
Sub InsertRows()
    'Declare the string variable for the InputBox entry.
    Dim CountInsertRows As String
    'Define the String variable as the InputBox entry.
    CountInsertRows = InputBox( _
        "Enter the number of rows to be inserted:", _
        "Insert how many rows below the active cell?")
    'Verify that a number was entered.
    'The Val function returns the numbers contained in a string as a numeric value.
    If CountInsertRows = "" Or Val(CountInsertRows) < 1 Then Exit Sub
    'Insert as many rows as the number that was entered.
    'The Resize property returns a Range object based on the number of rows
    'and columns in the new range. The number that was entered in the InputBox
    'represents how many rows shall be inserted. The count of columns, which is
    'the other optional argument for Resize, need not be specified because it is
    'only rows being inserted.
    Rows(ActiveCell.Row + 1).Resize(Val(CountInsertRows)).Insert
End Sub
```

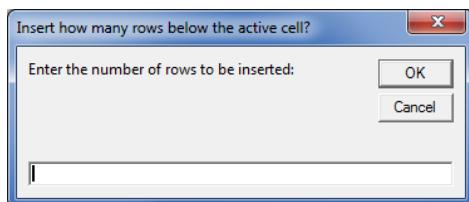


FIGURE 9-2

TRY IT

For this lesson, the active worksheet is currently protected with a password, and you ask the workbook's users if they want to unprotect the worksheet. If they answer No, the macro terminates. If they answer Yes, the macro proceeds to ask them for the password. If the

attempted password is incorrect, the user is informed of that, the worksheet remains protected, and the macro terminates. If the attempted password is correct, the user is then allowed to unprotect the worksheet.

Lesson Requirements

To get the sample workbook file you can download Lesson 9 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

It's a wise practice to ask the user to confirm her intention to proceed with the macro. There are plenty of instances when a user mistakenly clicks a button or triggers a macro that she had no intention of running.

In Step 5, `vbQuestion` adds a user-friendly touch of a question mark icon in your message boxes that ask the user a question.

Step 9 shows this example of the single line `If` statement:

```
If myPassword = "" Then Exit Sub
```

Some VBA programmers (me included) find that syntax more efficient than the following three-line syntax. Try both styles yourself and see what works best for you.

```
If myPassword = "" Then
Exit Sub
End If
```

Step-by-Step

1. Start by opening a new workbook and password protecting Sheet1 with the password `hello` (without quotes, all lowercase just as you see it here).
2. With your Sheet1 worksheet protected, press Alt+F11 to go to the Visual Basic Editor.
3. From the menu bar at the top of the VBE, click Insert \Rightarrow Module.
4. In the module you just created, type `Sub PasswordTest` and press Enter. VBA automatically places a pair of empty parentheses at the end of the sub line, followed by an empty line, and the `End Sub` line below that. Your macro should look like this so far:

```
Sub PasswordTest()
```

```
End Sub
```

5. Begin a `Select Case` structure with a Yes No Question message box to ask the users to confirm their intention to unprotect the worksheet:

```
Select Case MsgBox(
"Do you want to unprotect the worksheet?", _
vbYesNo + vbQuestion, _
"Please confirm your intentions.")
```

6. Handle the case for a No answer by informing the user that the macro will not continue, and then exit the macro with the `Exit Sub` statement:

```
Case vbNo  
MsgBox "No problem -- this macro will end.", vbInformation, "You clicked No."  
Exit Sub
```

7. Handle the case for a Yes answer:

```
Case vbYes
```

8. Provide an `InputBox` for the user to enter the password. Declare a `String` type variable, and define it as the text that will be entered into the `InputBox`:

```
Dim myPassword As String  
myPassword = _  
InputBox("Please enter the case-sensitive password:", _  
"A password is required to unprotect this worksheet.")
```

9. Here is an opportunity to add a single-line `If` statement to end the macro if the user clicks Cancel, or clicks OK without entering anything into the `InputBox`. The pair of double quotes with nothing between them is interpreted by VBA as a zero-length string:

```
If myPassword = "" Then Exit Sub
```

10. Begin an `If...Then` structure to determine if the `InputBox` entry matches the password `hello` that was used to protect the worksheet:

```
If myPassword <> "hello" Then
```

11. If the `InputBox` entry is anything other than `hello`, enter the code you would want to be executed when an incorrect password is entered, which you can do with a friendly message box:

```
MsgBox _  
"Sorry, " & myPassword & " is not the correct Password.", _  
vbCritical, _  
"Incorrect."
```

12. Enter your `Else` statement and supply the code to be executed only if the correct password is entered:

```
Else  
MsgBox _  
"Thank you. Please click OK to unprotect the worksheet.", _  
vbInformation, _  
"You entered the correct password!!"  
ActiveSheet.Unprotect "hello"
```

13. End the `If` structure that determined if the `InputBox` entry matched the password `hello`:

```
End If
```

14. End the `Select Case` structure for the users to confirm their intention of unprotecting the worksheet:

```
End Select
```

15. Here is what the complete macro would look like:

```

Sub PasswordTest()
'Ask the user if they want to unprotect the worksheet.
Select Case MsgBox( _
"Do you want to unprotect the worksheet?", _
vbYesNo + vbQuestion, _
"Please confirm your intentions.")
'Handle the case for a No answer by informing the user
'that the macro will not continue,
'and then exit the subroutine with the Exit Sub statement.
Case vbNo
MsgBox "No problem -- this macro will end.", vbInformation, "You clicked No."
Exit Sub
'Handle the case for a Yes answer by providing an InputBox
'for the user to enter the password.
Case vbYes
'Declare a String type variable.
Dim myPassword As String
'Define the String variable as the text that will be entered into the InputBox.
myPassword =
InputBox("Please enter the case-sensitive password:", _
"A password is required to unprotect this worksheet.")
'A one-line If statement to end the macro if the user clicks Cancel,
'or clicks OK without entering anything into the InputBox.
If myPassword = "" Then Exit Sub
'If structure to determine if the InputBox entry matches the password "hello"
'that was used to protect the worksheet.
If myPassword <> "hello" Then
'The code line to be executed if an incorrect password is entered.
MsgBox _
"Sorry, " & myPassword & " is not the correct Password.", _
vbCritical, _
"Incorrect."
Else
'The code to execute only if the correct password is entered.
MsgBox _
"Thank you. Please click OK to unprotect the worksheet.", _
vbInformation, _
"You entered the correct password!!"
ActiveSheet.Unprotect "hello"
'End the If structure that determined if the InputBox entry
'matched the password "hello".
End If
'End the Select Case structure for the users to confirm their intention
'of unprotecting the worksheet.
End Select
End Sub

```

REFERENCE Please select the video for Lesson 9 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

PART III

Beyond the Macro Recorder: Writing Your Own Code

- ▶ **LESSON 10:** Repeating Actions with Loops
- ▶ **LESSON 11:** Programming Formulas
- ▶ **LESSON 12:** Working with Arrays
- ▶ **LESSON 13:** Automating Procedures with Worksheet Events
- ▶ **LESSON 14:** Automating Procedures with Workbook Events
- ▶ **LESSON 15:** Handling Duplicate Items and Records
- ▶ **LESSON 16:** Using Embedded Controls
- ▶ **LESSON 17:** Programming Charts
- ▶ **LESSON 18:** Programming PivotTables and PivotCharts
- ▶ **LESSON 19:** User-Defined Functions
- ▶ **LESSON 20:** Debugging Your Code

10

Repeating Actions with Loops

Suppose you need to perform the same action, or the same sequence of several actions, many times in your macro. For example, you may need to unhide all worksheets that are hidden, or you need to add 12 worksheets to your workbook and name them for each month of the year.

The fact is, you'll encounter many circumstances for which a repetition of similar commands is a necessary part of the job. In most cases it will be impractical, and sometimes downright impossible, to write an individual command for each performance of the action. The need for handling a repetitive set of commands efficiently is exactly what loops are made for.

WHAT IS A LOOP?

A *loop* is a method of performing a task more than once. You may need to copy each worksheet in your workbook and save it as the only worksheet in its own separate workbook. Or, you may have a list of thousands of records and you want to insert an empty row where the value of a cell in column A is different than the value of the cell below it. Maybe your worksheet has dozens of cells that contain comments, and you want to add the same preceding text to every comment's existing text without having to edit every comment one at a time.

Instead of doing these kinds of tasks manually, or recording an impractical (and sometimes impossible) macro to handle the repetition, you can use loops to get the job done with less code while keeping more flexible control over the number of necessary repetitions. In VBA, a loop is a structure that executes one or more commands, and then cycles through the process again within the structure, for as many times as you specify. Each cycle of executing the loop structure's command(s) is called an *iteration*.

NOTE *Loops are great, but you're not obligated to use one just because you need to repeat an action two or three times. You'll come across situations that you know will always require the same commands to be repeated the same way, for the same number of times. If you feel like coding each action separately, and you can live with the longer code, go ahead and hard-code the separate commands if that's what works for you. Beyond three potential iterations, however, you really should go the loop route. It'll save you a lot of work, and the code will be easier to maintain.*

The number of a loop's iterations depends on the nature of the task at hand. All loops fall into one of two categories. A *fixed-iteration* loop executes a specified number of times that you hard-code directly as a numeric expression. An *indefinite loop* executes a flexible number of times that is usually defined by a logical expression.

For example, a fixed iteration loop dealing with a year's worth of data might need to cycle through 12 iterations, one for each month. An indefinite loop might need to cycle through every worksheet in your workbook, taking into consideration that because worksheets can be added or deleted at any time, the exact count of worksheets can never be known in advance.

Types of Loops

VBA provides several different looping structures, and at least one of them will be suited for any looping requirement you'll encounter. Table 10-1 shows an overview of the types of loops in VBA.

TABLE 10-1: Types of Loops in VBA

LOOP STRUCTURE	CATEGORY	EXPLANATION
For...Next	Fixed	Repeats an action for a specified number of times.
For Each...Next	Fixed	Repeats an action upon an object in a Collection. For example, you can perform a task for each worksheet in the workbook.
Do While	Indefinite	Executes an action if the condition is True, and repeats the action until the condition is False.
Do Until	Indefinite	Executes an action if the condition is False, and repeats the action until the condition is True.
Do...Loop While	Indefinite	Executes an action once, and repeats the action while the condition is True, until it is False.
Do...Loop Until	Indefinite	Executes an action once, and repeats the action while the condition is False, until it is True.
While...Wend	Indefinite	Same as the Do While loop structure, still supported by VBA but obsolete.

For...Next

The For...Next loop structure is a simple and effective way to repeat an action for a specified number of times. For example, if you want to add five new worksheets to your workbook, you could declare an Integer type variable and repeat the action five times, like this:

```
Sub AddFiveWorksheets()
    'Declare your Integer or Long variable.
    Dim intCounter As Integer
    'Open the For loop structure.
```

```

For intCounter = 1 To 5
'Enter the command(s)that will be repeated.
Worksheets.Add
'Loop to the next iteration.
Next intCounter
End Sub

```

NOTE Although it is technically correct that the Next statement can stand alone, do yourself a favor by getting into the good habit of including the variable in the Next statement. For example, writing your code as Next intCounter instead of just as Next makes it easier for you to read and for other people to understand.

When VBA executes a For...Next loop, by default it increments by 1 the value of the declared Integer or Long type variable. Because the objective was to add five worksheets, the easiest way to keep a running count of the process is to iterate five times, just as if you were counting the occurrence of each action from 1 to 5.

You can take advantage of the fixed nature of a For...Next loop by asking for the number of worksheets that are to be added. In the following example, an InputBox engages the user by asking for a number that represents how many worksheets will be added:

```

Sub ForNextExample2()
'Declare your Integer or Long variables.
Dim MoreSheets As Integer, intCounter As Integer
'Define the MoreSheets variable with an InputBox.
MoreSheets = InputBox( _
"How many worksheets do you want to add?", _
"Enter a number")
'Open the For loop structure.
For intCounter = 1 To MoreSheets
'Enter the command(s)that will be repeated.
Worksheets.Add
'Loop to the next iteration.
Next intCounter
End Sub

```

You don't always need to start counting from the number 1 in a For...Next loop; you can pretty much count from any number to any number. Suppose you want to hide rows 6, 7, and 8. A For...Next loop to accomplish that task could look like this:

```

Sub ForNextExample3()
'Declare your Integer or Long variable.
Dim intCounter As Integer
'Open the For loop structure.
For intCounter = 6 To 8
'Enter the command(s)that will be repeated.
Rows(intCounter).Hidden = True
'Loop to the next iteration.
Next intCounter
End Sub

```

For Each...Next

The `For Each...Next` loop executes an action for a fixed number of times just as the `For...Next` construct does, but unlike `For...Next`, `For Each...Next` does not keep a count along the way of how many iterations it performs. The count of iterations is not important with `For Each...Next` because the objective is to execute an action for however many objects exist in a specified VBA collection. Maybe there will be hundreds of iterations to occur; maybe there will be none.

Suppose that as part of your workbook project's design, a particularly lengthy macro will run faster and less confusingly for the user if all other Excel workbooks are closed. Naturally, you can never know in advance whether the user will have 10 other workbooks open in addition to yours, or whether your workbook is the only open workbook. A `For Each...Next` loop would be the perfect way to save and close all other workbooks that might be open, such as with this example:

```
Sub CloseWorkbooks()
    'Declare your object variable.
    Dim wb As Workbook
    'Open the For loop structure.
    For Each wb In Workbooks
        'Enter the command(s) that will be repeated.
        If wb.Name <> ThisWorkbook.Name Then
            wb.Save
            wb.Close
        End If
        'Loop to the next iteration.
    Next wb
End Sub
```

Notice that an object variable is declared for `Workbook`, and the `Workbooks` collection is being evaluated with an `If` structure for the presence of any and all workbooks that are named differently than your workbook. The code will complete its mission with the same result of your workbook being the only one that's open, regardless of whether it was the only one open from the start, or whether 50 other workbooks had also been open at the time.

One of Excel's oddities is that you can hide any number of worksheets at the same time, but if you have multiple worksheets that are hidden, you can unhide only one worksheet at a time. With this macro as another example of a `For Each...Next` loop, you can quickly unhide all worksheets at once:

```
Sub UnhideSheets()
    'Declare your object variable.
    Dim ws As Worksheet
    'Open a For Each...Next loop.
    For Each ws In Worksheets
        'Command(s) to be executed.
        ws.Visible = xlSheetVisible
        'Loop to the next iteration.
    Next ws
End Sub
```

Exiting a For...Loop

Suppose your macro requires that you determine whether a particular workbook named `Test.xlsx` happens to be open, and if so, you must close it. You might compose a macro with a loop that looks like this:

```

Sub CloseOneWorkbook()
'Declare your object variable.
Dim wb As Workbook
'Open a For Each loop.
For Each wb In Workbooks
'Command(s) to be executed.
If wb.Name = "Test.xlsx" Then
wb.Save
wb.Close
End If
'Loop to the next iteration.
Next wb
End Sub

```

Strictly speaking, the macro works. But think for a moment—what if a few dozen workbooks are open? In this case, you'd want the loop to do its job only up to the point of encountering the Test.xlsx workbook.

In the preceding `CloseOneWorkbook` example, even if the `Test.xlsx` workbook is found to be open and then closed, the loop still continues its appointed rounds after that by unnecessarily evaluating each open workbook. This would be a waste of time and system resources. Instead, you should insert the `Exit For` statement to stop the looping process in a `For...Next` or `For Each...Next` loop when a condition has been met and dealt with, and cannot be met thereafter.

Here is an example of how that macro should look, with the `Exit For` statement placed immediately before the `End If` statement:

```

Sub CloseOneWorkbookFaster()
'Declare your object variable.
Dim wb As Workbook
For Each wb In Workbooks
'Command(s) to be executed.
If wb.Name = "Test.xlsx" Then
wb.Save
wb.Close
'Exit For statement to avoid needless iterations if the condition is met.
Exit For
End If
'Loop to the next iteration.
Next wb
End Sub

```

Looping in Reverse with Step

A common request that Excel users have is to insert an empty row when the value of a cell in some particular column does not equal the value of the cell below it. In Figure 10-1, the table of data is sorted by Region in column A, and the request is to visually separate the regions with an empty row at each change in Region name.

When inserting a series of rows like this, it's best to start looping from the bottom of the table, and work your way up to the top. That means your numeric row reference in the loop will be decreasing and not increasing, because your starting point is row 18 (the last row of data) and your ending point is row 2 (the first row of data).

Before				After				
	A	B	C	D	A	B	C	D
1	Region	Item	Count		1	Region	Item	Count
2	East	Wombats	116		2	East	Wombats	116
3	East	Widgets	654		3	East	Widgets	654
4	East	Wallabees	822		4	East	Wallabees	822
5	East	Wallabees	456		5	East	Wallabees	456
6	East	Wombats	898		6	East	Wombats	898
7	West	Wallabees	605		7			
8	West	Witches	781		8	West	Wallabees	605
9	West	Wallabees	990		9	West	Witches	781
10	North	Wallabees	349		10	West	Wallabees	990
11	North	Wombats	493		11			
12	North	Widgets	507		12	North	Wallabees	349
13	North	Widgets	644		13	North	Wombats	493
14	South	Widgets	570		14	North	Widgets	507
15	South	Widgets	323		15	North	Widgets	644
16	South	Wallabees	373		16			
17	South	Wallabees	900		17	South	Widgets	570
18	South	Widgets	962		18	South	Widgets	323
19					19	South	Wallabees	373
20					20	South	Wallabees	900
21					21	South	Widgets	962

FIGURE 10-1

Recall that when VBA executes a `For` `Next` loop, by default it increments by 1 the value of your declared `Integer` or `Long` type variable. With `For...Next` loops, you can specify an alternative increment or decrement value by using the optional `Step` keyword. You can step forward or backward by as large a numeric value as you like.

In this example, each cell in column A is being evaluated one by one, from row 18 to row 2, so the loop will step by a numeric factor of negative 1. Here is a macro that makes the “Before” image look like the “After” image in Figure 10-1:

```
Sub InsertRows()
    'Declare your Integer or Long variable.
    Dim xRow As Long
    'Open a For Each loop.
    For xRow = 18 To 3 Step -1
        'Command(s) to be executed.
        If Range("A" & xRow).Value <> Range("A" & xRow - 1) Then
            Rows(xRow).Resize(1).Insert
        End If
        'Loop to the next iteration.
    Next xRow
End Sub
```

Do While

The `Do` statement is an extremely powerful tool with which to gain more flexibility in your looping structures. In a `Do While` loop, you test for a condition that must be True before the loop will execute. When the condition is True, the command(s) within the loop are executed.

As a simple example, the `DoWhileExample` macro produces five message boxes because the `Do While` loop tests for the condition that an `Integer` variable (named `iCounter`) has not exceeded the number 5. Notice that the `iCounter` variable starts at 1 outside the loop and is increased by 1 inside the loop:

```
Sub DoWhileExample()
    Dim iCounter As Integer
    iCounter = 1
    Do While iCounter <= 5
        MsgBox "Hello world!", , iCounter
        iCounter = iCounter + 1
    Loop
End Sub
```

Let's apply this concept to a more practical activity; suppose you want to open all Excel workbooks that are in a particular file path. The macro named `OpenAllFiles` does that using a `Do Loop` structure. The `Dir` function returns the first filename that matches the combination of the specified pathname and an Excel workbook extension containing `.xls`. Calling the `Dir` function again opens additional filenames until a filename is encountered that does not match the combination:

```
Sub OpenAllFiles()
    Dim myFile As String, myPath As String
    myPath = "C:\Your File Path\
    myFile = Dir(myPath & "*.*")
    Do While myFile <> ""
        Workbooks.Open myPath & myFile
        myFile = Dir()
    Loop
End Sub
```

Do Until

When VBA runs a `Do Until` loop, it tests the logical condition you supply and executes the commands within the loop as long as the condition evaluates to `False`. When VBA reaches the `Loop` statement, it re-evaluates the condition and executes the looping commands only if the condition is still `False`.

This example demonstrates `Do Until` by selecting the next worksheet based on the index number from whatever current worksheet you are on. The wrinkle that is taken into consideration by the loop is that the next highest index number worksheet might be hidden, and because you cannot select a hidden worksheet, the loop selects the next highest index number of a worksheet that is also visible:

```
Sub SelectSheet()
    'Declare an Integer type variable to handle the Index number property
    'of whichever worksheet(s) are being evaluated in the current iteration.
    Dim intWS As Integer
    'Because you want to activate the next visible worksheet,
    'as a starting point you need to know the next highest Index position
    'from whatever worksheet is active at the time.
    intWS = ActiveSheet.Index + 1
    'If you are on the last worksheet, you'll have reached the end of the line,
    'so define the intWS as the first Index worksheet.
    If intWS>Worksheets.Count Then intWS = 1
    'Open a Do Until loop that determines the next Index number,
    'only considering visible worksheets.
```

```
Do Until Worksheets(intWS).Visible = True
    'Add a 1 to the intWS variable as you iterate to the next highest Index number.
    intWS = intWS + 1
    'If it turns out that the intWS Index variable reaches a number
    'that is greater than the count of worksheets in the workbook,
    'the intWS number is set back to 1, which is the first Index position
    If intWS > Worksheets.Count Then intWS = 1
    'Loop to start evaluation again, until the proper Index number is found.
    Loop
    'Select the worksheet whose Index property matches the index number
    'that has met all the criteria.
    Worksheets(intWS).Select
End Sub
```

For another example, suppose you want to update your AutoCorrect list easily and quickly. Say you have a two-column table on your worksheet that occupies columns A and B. In column A, you have listed frequently misspelled words, and in column B are the corrected words that you want Excel to automatically display if you misspell any of those words. For example, in cell A1 you have entered `teh` and in cell B1 you have entered the correction of `the`. The following macro uses a `Do Until` loop to handle each entry in column A and continues to do so until the first empty cell is encountered, indicating the end of the list:

```
Sub AddCorrection()
    'Declare a Long type variable to help looping through rows
    'of the two-column list.
    Dim i As Long
    'Declare two String type variables:
    'one for the original entry, and the other for the text string replacement.
    Dim myMistake As String, myCorrection As String
    'Establish the number 1 for the Long Variable, representing row 1
    'which is the first row in the example list.
    i = 1
    'Open a Do Until loop, telling VBA to stop looping when an empty cell
    'is encountered in column A, indicating the end of the list.
    Do Until IsEmpty(Cells(i, 1))
        'Define the myMistake variable as the text contents of the cell in column A
        myMistake = Cells(i, 1).Value
        'Define the myCorrection variable as the text contents of the cell in column B.
        myCorrection = Cells(i, 2).Value
        'VBA tells the Excel Application's AutoCorrect property to update itself with
        'the two strings from columns A and B.
        Application.AutoCorrect.AddReplacement What:=myMistake, Replacement:=myCorrection
        'Add a 1 to the i variable in preparation for evaluating the next row in the list.
        i = i + 1
        'The Loop statement starts the process again for the next row in the list.
    Loop
End Sub
```

NOTE This example utilizes the `Cells` range method, which to some VBA newcomers can take a little getting used to. If you need a reminder for the use of `Cells`, an explanation is in Lesson 8, in the section “Using the `Cells` Property.” You’ll be seeing an increased use of the `Cells` method in this book because it is such an easier and more efficient method of referring to dynamic ranges in VBA.

Do...Loop While

To have VBA test the conditional statement after executing the commands within the loop, you simply place the conditional statement after the `Loop` keyword. The `Do...Loop While` syntax is

```
Do
    Command statements to be executed within the loop.
Loop While condition
```

When VBA executes the command(s) in a `Do...Loop While` structure, it does so first, and then at the `Loop While` line, it tests the logical condition. If the condition is True at that point, the loop iterates again, and so on, until the condition evaluates to False.

A common request is to locate all cells in a worksheet that contain a particular value, similar to clicking the Find Next button on the Find dialog box, and then do something to that cell or to the cells around it. Suppose you have a worksheet filled with data and you want to find all cells that contain the word “Hello.” These cells can be in any row or column.

For each of those cells where “Hello” is found, you want to place the word “Goodbye” in the cell of the column to the immediate right. The following macro does just that using a `Do...Loop While` construction that finds every cell containing “Hello” and identifies its address, so the loop can perform only as many iterations as there are cells containing “Hello”:

```
Sub FindHello()
    Dim HelloCell As Range, BeginningAddress As String
    Set HelloCell = ActiveSheet.UsedRange.Find("Hello", LookIn:=xlValues)
    If Not HelloCell Is Nothing Then
        BeginningAddress = HelloCell.Address
        Do
            HelloCell.Offset(0, 1).Value = "Goodbye"
            Set HelloCell = ActiveSheet.UsedRange.FindNext(HelloCell)
        Loop While Not HelloCell Is Nothing And HelloCell.Address <> BeginningAddress
    End If
End Sub
```

Do...Loop Until

Similar in approach to the `Do...Loop While` construct, the `Do...Loop Until` loop tests its condition after executing the loop’s statements. The `Until` keyword tells VBA that the statements within the loop will be executed again for as long as the logical condition evaluates to False. After VBA tests the condition as True, the loop’s iterations stop, and the macro resumes with the line of code following the `Loop` keyword.

This macro shows an example of a `Do...Loop Until` structure, which creates 365 new worksheets, all named with dates starting from the day you run the macro:

```
Sub YearSheets()
    Dim i As Integer
    i = 0
    Do
        Sheets.Add(After:=Sheets(Sheets.Count)).Name = Format(VBA.Date + i, "MM-DD-YYYY")
        i = i + 1
    Loop Until i = 365
End Sub
```

While...Wend

While...Wend loops have become obsolete and are rarely used because they are not as robust as Do and For loops. VBA still supports While...Wend loops for backward compatibility with prior versions of Excel, and I am not aware of any plans by Microsoft to stop supporting While...Wend.

So, though I recommend you not bother learning how to build a While...Wend loop, the fact is, they are rather uncomplicated constructs and you should have some familiarity with how they look if you should see them in code written by others. Here is an example of While...Wend that uses an InputBox that asks for a password, and keeps asking until the correct password is entered, or the message box is canceled:

```
Sub InputPassword()
    While InputBox("Please enter password:", "Password required") < > "MyPassword"
        If MsgBox( _
            "Sorry, that is not correct.", _
            vbOKCancel, _
            "Wrong password") _
            = vbCancel Then End
    Wend
    MsgBox "Yes!! You entered the correct password!", vbOKOnly, "Thank you!"
End Sub
```

NESTING LOOPS

Your macros will eventually require that you enclose one loop structure inside another loop structure, referred to as *nesting loops*. For example, you may need to loop through a set of rows in a data table, and each completed set of looped-through rows will represent a single iteration for a larger loop construct for the columns in the table.

When you nest loops, you need to be aware of a few important points:

- When you nest For...Next loops, each loop must have its own uniquely named counter variable.
- When you nest For Each...Next loops, each loop must have its own uniquely named object (or element) variable.
- If you use an Exit For or Exit Do statement, only the loop that is currently executing will terminate. If that loop is nested within a larger loop, the larger loop still continues to execute its iterations.
- I mentioned it earlier in this lesson, but it especially holds true with nested loops: I strongly recommend you include the variable name in your Next statements.

Here is an example of a macro with a Do loop nested inside a For Each...Next loop. The following macro produces a list of six unique random numbers between 1 and 54, similar to a lottery drawing:

```
Sub PickSixLottery()
    'Declare the Range variables for the entire six-cell range,
    'and for each individual cell in the six-cell range.
    Dim RandomRange As Range, RandomCell As Range
```

```

'Identify the six-cell range where the randomly selected numbers will be listed.
Set RandomRange = Range("A1:A6")
'Before populating the six-cell list range, make sure all its cells are empty.
RandomRange.Clear
'Open a For...Each loop to cycle through each cell in range A1:A6.
For Each RandomCell In RandomRange
    'Open a Do...Loop that enters a unique random number between 1 and 54
    Do
        RandomCell.Value = Int(54 * Rnd + 1)
    Loop Until WorksheetFunction.CountIf(RandomRange, RandomCell.Value) = 1
    'Iterate to the next cell until all six cells have been populated.
    Next RandomCell
End Sub

```

TRY IT

For this lesson, you write a macro that uses a `For...Next` loop with an `Integer` type variable that adds 12 worksheets to your workbook, names each worksheet by calendar month (“January,” “February,” and so on), and places the worksheets’ tabs in order of calendar month from left to right.

Lesson Requirements

To get the sample workbook file you can download Lesson 10 from the book’s website at www.wrox.com/go/excelvba24hour.

Hints

In Step 6, the `DateSerial` function requires three arguments, in the sequence of the year, the month, and the day. When the task at hand is to list the 12 calendar months, any year number will do. The day should be a basic number every month has, making the number 1 a good choice.

This macro adds new worksheets with month names. Running the macro again without deleting the same worksheets it created would cause the macro to error because a workbook cannot contain duplicate worksheet names. Lesson 20 shows how to handle errors.

Step-by-Step

1. Open a new workbook and press Alt+F11 to go to the Visual Basic Editor.
2. From the menu bar at the top of the VBE, select Insert \leftrightarrow Module.
3. In the module you just created, type `Sub LoopTwelveMonths` and press Enter. VBA automatically places a pair of empty parentheses at the end of the `Sub` line, followed by an empty line, and the `End Sub` line below that. Your macro looks like this so far:

```
Sub LoopTwelveMonths ()
```

```
End Sub
```

4. Declare an `Integer` type variable that iterates 12 times, one for each month of the year:

```
Dim intMonth As Integer
```

5. Open a For...Next loop that starts from 1 and ends at 12:

```
For intMonth = 1 To 12
```

6. With a one-line command, you can add each of the 12 worksheets in turn, while placing their tabs one after another from left to right, and naming each tab by calendar month. The DateSerial function is a good way to cycle through month names because it requires integer values for the arguments of Year, Month, and Day, just like the DATE worksheet function. You can use any year, and any day that is not a number greater than 28. For the Month argument, the intMonth variable is a perfect fit because it was declared as an Integer type:

```
Sheets.Add(After:=Sheets(Sheets.Count)).Name = _  
Format(DateSerial(2011, intMonth, 1), "MMMM")
```

7. Enter the Next statement for the intMonth variable that produces and names the next month's worksheet up to and including December:

```
Next intMonth
```

8. When completed, the macro looks like this, with comments that have been added to explain each step:

```
Sub LoopTwelveMonths()  
'Declare an Integer type variable to iterate twelve times,  
'one for each month of the year.  
Dim intMonth As Integer  
'Open a For...Next loop that starts from one and ends at twelve.  
For intMonth = 1 To 12  
'With a one-line command, you can add each of the twelve worksheets in turn,  
'while placing their tabs one after another from left to right.  
Sheets.Add(After:=Sheets(Sheets.Count)).Name = _  
Format(DateSerial(2011, intMonth, 1), "MMMM")  
'The Next statement for the intMonth variable  
'produces and names the next month worksheet.  
Next intMonth  
End Sub
```

REFERENCE Please select the video for Lesson 10 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

11

Programming Formulas

Spreadsheets are a popular choice for managing information because mathematical calculations and data analysis are, and always will be, a requirement of education, business, and personal record-keeping. If there were no need to compile numeric data with formulas, there'd be no need for spreadsheets as we know them—an unfathomable thought in our information-ravenous, digital world.

As you've seen, VBA enables you to programmatically manipulate Excel's objects, methods, and properties. You can interact with users to make decisions and establish conditions. Just as importantly, you need to understand how to program formulas, starting with how Excel regards locations of cells and ranges by their row and column references.

UNDERSTANDING A1 AND R1C1 REFERENCES

Most people who use Excel—most being around 99.9 percent—view Excel worksheets with rows headed from the top as numbers 1, 2, 3 and continuing downward, and columns headed from the left as letters A, B, C and continuing to the right. The top-left cell address on the Excel grid is commonly seen as cell A1. The cell immediately below A1 is A2, the cell to the right of A2 is B2, and so on.

Behind the scenes, Excel does not refer to its rows and columns in *A1 style*; that is, not in the sequence of column letter and row number. Rather, Excel regards rows and columns as numbers, in *R1C1 style*, expressing a cell address in the sequence of its intersecting row number and column number.

NOTE *If you are wondering if understanding R1C1 style is really important enough to stay with this lesson, the answer is yes, it really is important enough. As concepts go, understanding R1C1 style gives you the most bang for your buck in terms of the long-term benefit you get from spending the few minutes to read this lesson. Your VBA programming skills will advance much faster and easier once you get a handle on R1C1 references.*

In R1C1 style, “R” stands for row and “C” stands for column. For example, cell D7 is identified by Excel as the address at the intersection of row 7 and column 4 (because column D is the fourth column from the left on the worksheet grid), which Excel interprets as R7C4. Cell M92 is interpreted as R92C13, and so on. As you might guess, the R1C1 address of cell A1 is R1C1.

Getting Started with a Few One-Liners

As you will see on the following pages, R1C1 cell references do not always look as clean as just a number for a row and a number for a column. The R1C1 style uses a starting reference point, and without one specified, assumes cell A1 as the default reference. For example, suppose your active cell is H22. To refer to a cell 3 rows up and 5 columns to the right, which is cell M19, then with cell H22 as the reference point (that is, as far as cell H22 is concerned), cell M19 would be referred to as =R[-3]C[5].

NOTE You can plug the following four examples of single code lines into a macro, or you can quickly execute them in the Immediate window. You may recall from Lesson 3 that you can access the Immediate window easily by pressing Alt+F11 to go to the Visual Basic Editor, then pressing Ctrl+G to enter the Immediate window. Just copy and paste any of these single code lines into the Immediate window and press Enter. To see the results, press Alt+Q to return to the worksheet.

To enter a formula programmatically in cell H22 that shows the value in cell M19, your line of code would be this, using a relative reference to cell M19:

```
Range("H22").FormulaR1C1 = "=R[-3]C[5]"
```

As another example, if you want the formula in cell H22 to return the value in cell H26, which is 4 rows greater than row number 22 and in the same column H, that code line would be as follows:

```
Range("H22").FormulaR1C1 = "=R[4]C"
```

If you want the formula in cell H22 to return the value in cell A22, which is on the same row but 7 columns less than column number 8 (Excel and VBA regard column H as column 8), that code line would be:

```
Range("H22").FormulaR1C1 = "=RC[-7]"
```

Finally, if you want to enter a formula in cell H22, or any cell for that matter, to return the value in cell F3, such that the formula’s row and column references are absolute (making the formula look like =\$F\$3), this line of code would do that:

```
Range("H22").FormulaR1C1 = "=R3C6"
```

NOTE You have probably noticed that Excel and VBA rarely regard cell addresses by column letter and row number. It can be a challenge at first to stray from the familiar thought process of referencing cell addresses in A1 style, considering the alpha column headers and worksheet formulas that are almost always how worksheets are viewed. Recall from Lesson 8 that the Cells property refers to addresses in R1C1 style, by their row number and column number. For example, the statement Cells(5, 2).Select would select cell B5 of the active worksheet, which is a syntax you have already seen. The more you work with VBA, the more you will see how useful the R1C1 style is, and how limiting the A1 style will be.

Comparing the Interface of A1 and R1C1 Styles

It's been said that a picture is worth a thousand words. Take a look at the next several figures to see worksheets from an R1C1 point of view. The comparison figures help you to see formulas and worksheets the way Excel and VBA sees them.

NOTE There's a pro-R1C1 tone to this chapter, but I'm not suggesting you change your worksheet viewing habits to the R1C1 view if you've been working in A1 view. In fact, I always work in A1 view, just as most Excel users do. The goal in this chapter is to explain what R1C1 is and how it works.

Figure 11-1 shows a side-by-side comparison of A1 and R1C1 styles for the same spreadsheet. Notice the active cell address in the name box, the column headers, and the formula as displayed in the formula bar vary between the two styles.

The figure consists of two side-by-side screenshots of an Excel spreadsheet. The left screenshot is labeled "You usually see this view: Workbook set to the default A1 style." and the right one is labeled "Sometimes you encounter this view: Workbook set to the R1C1 style." Both screenshots show a table with columns "Salesperson" and "Widgets sold".

- A1 Style (Left):**
 - Active cell: B10
 - Name box: =SUM(B4:B9)
 - Column headers: A, B, C, D, E
 - Formula bar: =SUM(B4:B9)
 - Cell B10 contains the value 29.
- R1C1 Style (Right):**
 - Active cell: R10C2
 - Name box: =SUM(R[-6]C:R[-1]C)
 - Column headers: 1, 2, 3, 4, 5
 - Formula bar: =SUM(R[-6]C:R[-1]C)
 - Cell R10C2 contains the value 29.

Annotations highlight differences:

- A yellow box with a red arrow points from the A1 style formula bar to the R1C1 style formula bar, with the text: "The name box displays the active cell address in R1C1 style."
- A yellow box with a red arrow points from the R1C1 style formula bar to the R1C1 style cell value, with the text: "Example of a relative reference formula in R1C1 style."
- A yellow box with a red arrow points from the R1C1 style cell value back to the R1C1 style formula bar, with the text: "Column headers are displayed as numbers."

You usually see this view: Workbook set to the default A1 style.		Sometimes you encounter this view: Workbook set to the R1C1 style.	
<p>B10</p> <p>1 Sales of Widgets this month</p> <p>2</p> <p>3 Salesperson Widgets sold</p> <p>4 Emily 9</p> <p>5 Andy 4</p> <p>6 Yuki 3</p> <p>7 Buddy 2</p> <p>8 Ruth 8</p> <p>9 Dylan 3</p> <p>10 Total 29</p>	<p>R10C2</p> <p>1 Sales of Widgets this month</p> <p>2</p> <p>3 Salesperson Widgets sold</p> <p>4 Emily 9</p> <p>5 Andy 4</p> <p>6 Yuki 3</p> <p>7 Buddy 2</p> <p>8 Ruth 8</p> <p>9 Dylan 3</p> <p>10 Total 29</p>		

FIGURE 11-1

Toggling between A1 and R1C1 Style Views

Occasionally on Excel forums, or in e-mails I receive from people who follow my work, the question comes up about how and why their worksheets inexplicably show column headers as numbers. The reason is that someone unwittingly changed the view in that workbook from A1 to R1C1 style, and forgot how to undo the mysterious deed.

NOTE Your workbook doesn't need to be in R1C1 style to use `.FormulaR1C1` in your code. This is just an exercise to show how to get in and out of R1C1 style, and what that style looks like.

Here's how to toggle between the two views. Start by clicking the File tab so that you go to the backstage view. Click the Options item on the vertical menu, as shown in Figure 11-2 when using Excel version 2013.

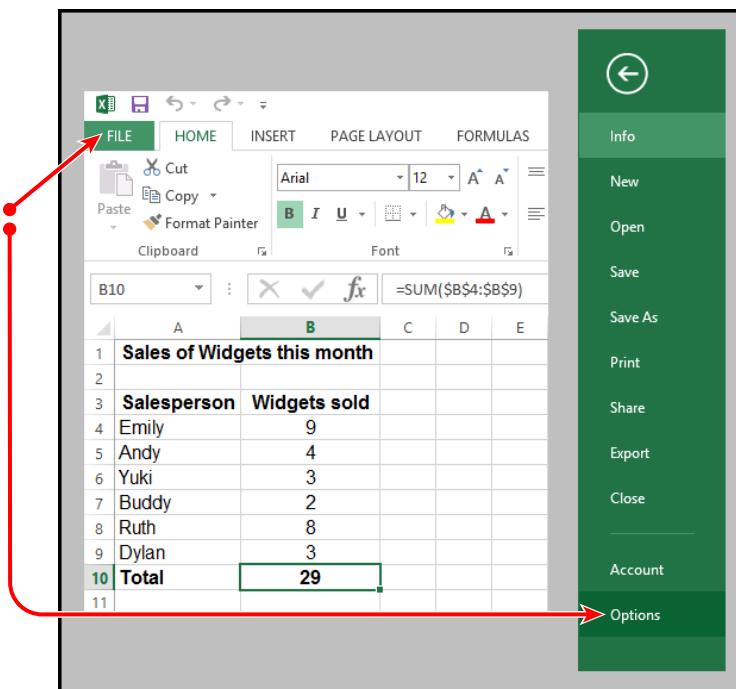


FIGURE 11-2

In the Excel Options dialog box, click the Formulas item in the menu pane at the left. Select the check box for R1C1 Reference Style in the Working with Formulas section, and click OK, as shown in Figure 11-3.

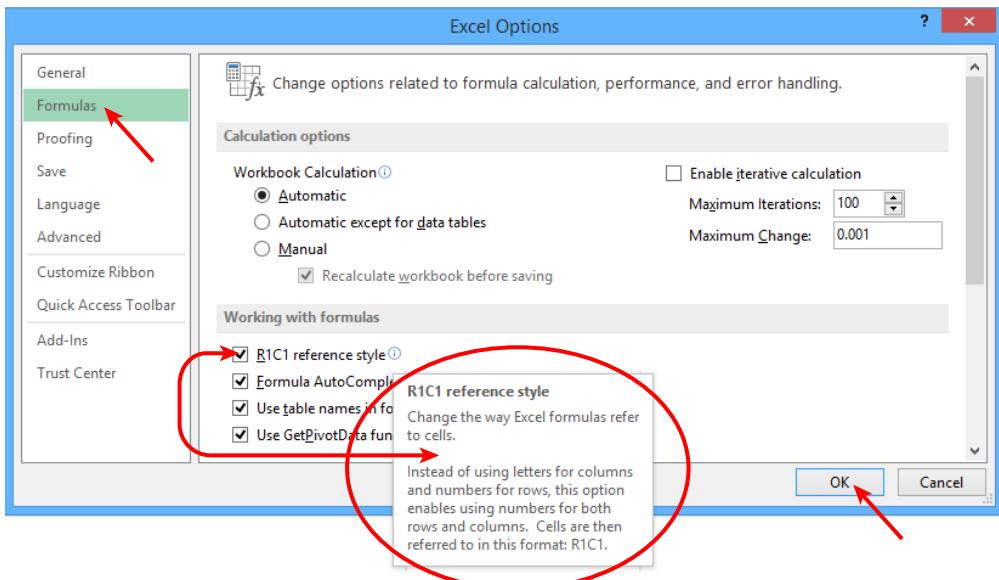


FIGURE 11-3

If you are using Excel 2003, click the Tools item on the menu bar, and select Options, as shown in Figure 11-4. In the Options dialog box, click the General tab, select R1C1 Reference Style in the Settings section, and click OK, as shown in Figure 11-5.

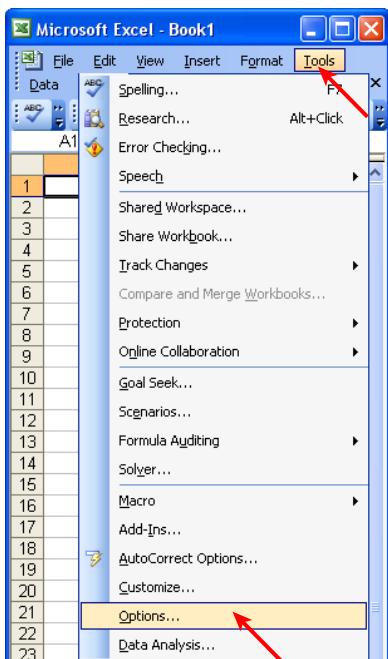


FIGURE 11-4

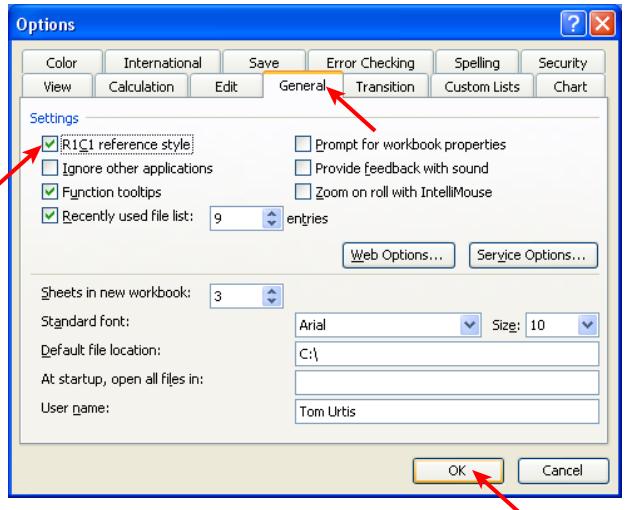


FIGURE 11-5

Here's another comparison of the two styles side by side. Figure 11-6 shows an example of an absolute reference formula in cell B10 (or if you prefer, in cell R10C2). To return to A1 style, simply repeat the preceding steps and deselect the option for R1C1 Reference Style.

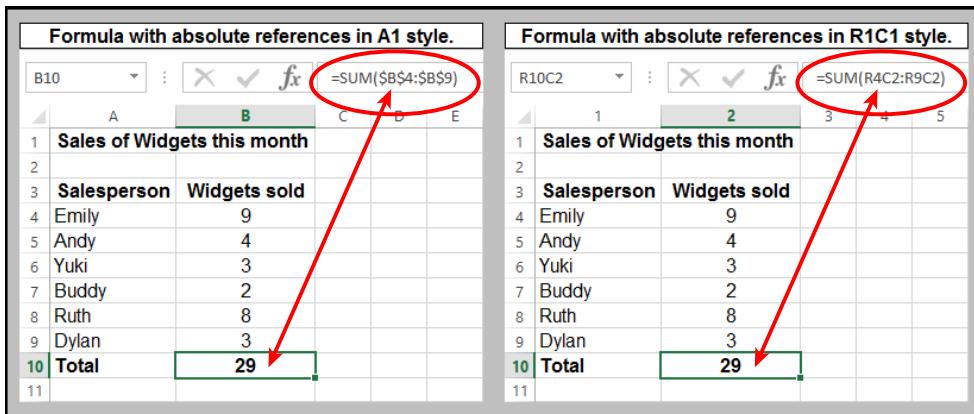


FIGURE 11-6

PROGRAMMING YOUR FORMULA SOLUTIONS WITH VBA

The following examples can give you some insight for designing formulas in macros to solve common situations. With VBA you can include variable names and named ranges in your formulas, providing creative ways to get your work done.

NOTE If and when you use the Macro Recorder to produce formulas to plug into your macros, you'll notice that formulas are recorded in R1C1 style, in whichever style you are in.

Using a Mixed Reference to Fill Empty Cells with the Value from Above

Figure 11-7 shows a before-and-after look at how you can use a mixed reference formula (as shown in the following snippet) to fill empty cells with the preceding constant value. Using the `SpecialCells` property, the same formula is entered into every blank cell in column A that is associated with the list:

```
Sub FillBlankCellsFromAbove()
    Application.ScreenUpdating = False
    With Columns(1)
        .SpecialCells(xlCellTypeBlanks).Formula = "=R[-1]C"
        'Convert formulas into static values.
        .Value = .Value
    End With
    Application.ScreenUpdating = True
End Sub
```

BEFORE		AFTER	
A	B	A	B
1 Salesperson	Items sold	1 Salesperson	Items sold
2 Ricky	Sweaters	2 Ricky	Sweaters
3	Jackets	3 Ricky	Jackets
4	Hats	4 Ricky	Hats
5 Jennifer	Jackets	5 Jennifer	Jackets
6	Skirts	6 Jennifer	Skirts
7	Shoes	7 Jennifer	Shoes
8	Dresses	8 Jennifer	Dresses
9	Belts	9 Jennifer	Belts
10 Willie	Socks	10 Willie	Socks
11	Shoes	11 Willie	Shoes
12 Sam	Jeans	12 Sam	Jeans
13	Coats	13 Sam	Coats
14	Scarves	14 Sam	Scarves
15	Shorts	15 Sam	Shorts

FIGURE 11-7

NOTE With Columns(1) refers to column A. If you were working with column H, you would have written the code as With Columns(8).

Using a Named Range with Relative, Mixed, and Absolute References

This example shows how to deal with several issues you might encounter. In Figure 11-8, a payroll worksheet needs a conditional formula in range D5:D12 to calculate the weekly salaries for each employee. Eligibility for overtime pay is based on the criteria of 40 maximum regular hours in cell B1. The overtime multiplication factor in cell B2 is the named range OvertimeFactor, which is multiplied for each hour past the 40-hour ceiling.

Regular Hours Maximum				
A	B	C	D	E
1 Regular Hours Maximum	40			
2 Overtime Increase Salary	1.5			
Name				
5 Sue Flay	42	\$ 30	\$ 1,290	
6 Brock Lee	40	\$ 28	\$ 1,120	
7 Carrie Oakey	36	\$ 32	\$ 1,152	
8 Jerry Atrick	46	\$ 35	\$ 1,715	
9 Mike Raffone	51	\$ 14	\$ 791	
10 Raynor Schein	13	\$ 26	\$ 338	
11 Sarah Bellum	47	\$ 31	\$ 1,566	
12 Ella Vader	32	\$ 42	\$ 1,344	
13				

FIGURE 11-8

The formula in cell D5 and copied to cell D12 is `=IF(B5<=B1,B5*C5,SUM((B5-B1)*OvertimeFactor,B1)*C5)`, which you can see in the formula bar in Figure 11-9. In the following macro, notice the syntax for relative, mixed, and absolute references, along with the inclusion of a named range, the `If` statement, and a nested `SUM` function:

```
Sub CalculateSalary()
    Range("D5:D12").FormulaR1C1 =
        "=IF(RC[-2]<=R1C2,RC[-2]*RC[-1],SUM((RC[-2]-R1C2)*OvertimeFactor,R1C2)*RC[-1])"
End Sub
```

	A	B	C	D	E
1	Regular Hours Maximum	40			
2	Overtime Increase Salary	1.5			
3					
4	Name	Hours Worked	Hourly Salary	Total Salary	
5	Sue Flay	42	\$ 30	\$ 1,290	
6	Brock Lee	40	\$ 28	\$ 1,120	
7	Carrie Oakey	36	\$ 32	\$ 1,152	
8	Jerry Atrick	46	\$ 35	\$ 1,715	
9	Mike Raffone	51	\$ 14	\$ 791	
10	Raynor Schein	13	\$ 26	\$ 338	
11	Sarah Bellum	47	\$ 31	\$ 1,566	
12	Ella Vader	32	\$ 42	\$ 1,344	
13					

FIGURE 11-9

Programming an Array Formula

As you know, when you compose an array formula manually, you must commit it to a worksheet cell by pressing the `Ctrl+Shift+Enter` keys, not just the `Enter` key. Similarly, when you want to install an array formula programmatically, you must use the `FormulaArray` method, not just the `Formula` or `Formula_R1C1` methods. The `FormulaArray` method is VBA's way of differentiating between an array and a non-array formula.

In Figure 11-10, array formulas are entered into destination cells B19, B20, and B21 with the following macro that averages scores for each of three lanes at a bowling alley. For a bit of variety to show an alternative cell reference syntax, I looped through each of the three destination cells (where the array formulas will go) using the `Range` statement that shows column letter B followed by the row numbers represented by a `Long` type variable named `lngRow`:

```
Sub AverageBowlingScores()
    Dim lngRow As Long
    For lngRow = 19 To 21
        Range("B" & lngRow).FormulaArray =
            "=AVERAGE(IF(R4C1:R16C1=RC[-1],R4C3:R16C6))"
    Next lngRow
End Sub
```

The screenshot shows an Excel spreadsheet with the following data:

Lane	Bowler	Score	Score	Score	Score
Lane 1	Emily	152	131	139	226
Lane 3	Yuki	158	122	195	227
Lane 3	Andy	189	111	182	144
Lane 2	Yuki	201	206	151	219
Lane 2	Emily	226	181	232	201
Lane 2	Yuki	130	159	132	200
Lane 3	Andy	221	147	164	110
Lane 1	Emily	287	237	170	224
Lane 1	Yuki	128	123	183	197
Lane 3	Andy	112	119	251	180
Lane 1	Emily	224	143	242	224
Lane 2	Yuki	225	108	263	210
Lane 3	Andy	181	167	163	163

Average scores by lane:

Lane 1	189.375
Lane 2	190.250
Lane 3	165.300

FIGURE 11-10

WARNING When you have a lot of formulas on a worksheet for which you want to convert all cell and range references from relative to absolute, this macro can do the job:

```
Sub ConvertRelativeToAbsolute()
    Dim cell As Range, strFormulaOld As String, strFormulaNew As String
    For Each cell In Cells.SpecialCells(xlCellTypeFormulas)
        strFormulaOld = cell.Formula
        strFormulaNew = _
            Application.ConvertFormula _
            (Formula:=strFormulaOld, fromReferenceStyle:=xlA1, _
            toReferenceStyle:=xlA1, toAbsolute:=xlAbsolute)
        cell.Formula = strFormulaNew
    Next cell
End Sub
```

And here's how you can convert absolute reference formulas to relative references on a worksheet:

```
Sub ConvertAbsoluteToRelative()
    Dim cell As Range, strFormulaOld As String, strFormulaNew As String
    For Each cell In Cells.SpecialCells(xlCellTypeFormulas)
        strFormulaOld = cell.Formula
        strFormulaNew = _
            Application.ConvertFormula _
            (Formula:=strFormulaOld, fromReferenceStyle:=xlA1, _
            toReferenceStyle:=xlA1, toAbsolute:=xlAbsolute)
        cell.Formula = WorksheetFunction.Substitute(strFormulaNew, "$", "")
    Next cell
End Sub
```

NOTE Here's a quick way to count your workbook's formulas, and show the total count in a message box subtotalled by worksheet name. The statements On Error Resume Next, Err.Number, and Err.Clear help to bypass potential stoppages of the macro, known as runtime errors, if (in this example) a worksheet does not contain any formulas. In Lesson 20, you become familiar with these and other error-related terms, along with techniques for handling errors in your code.

```
Sub CountFormulas()
Dim SheetFormulaCount As Long, TotalFormulaCount As Long
Dim myList As String, WS As Worksheet
SheetFormulaCount = 0: TotalFormulaCount = 0: myList = ""
For Each WS In Worksheets
'optional if your sheets are protected
'WS.Unprotect ("YourPassword")
On Error Resume Next
SheetFormulaCount = WS.Cells.SpecialCells(xlCellTypeFormulas).Count
If Err.Number <> 0 Then
Err.Clear
SheetFormulaCount = 0
End If
TotalFormulaCount = TotalFormulaCount + SheetFormulaCount
myList = myList & "Formula count in '" & WS.Name & "' : " & _
Format(SheetFormulaCount, "#,##0") & vbCrLf
'optional reprotect your sheets
'WS.Protect ("YourPassword")
Next WS
MsgBox myList & vbCrLf & "Total formulas in " & _
ThisWorkbook.Name & ":" & _
Format(TotalFormulaCount, "#,##0"), , "Workbook formula count"
End Sub
```

Summing Lists of Different Sizes along a Single Row

In Figure 11-11, a table has several columns, each containing a varying count of numeric entries needing to be summed. When you want to show the sums of each column along a single row, you first need to identify the last used row among all the columns, and install your sum formulas in the next row below that. The idea is to place the sums for each column in the first available row that has no data in any column.

	A	B	C	D
1	We R Widgets, year-to-date sales			
2	Jerry Atrick	Sue Flay	Carrie Okie	Ella Vader
3	\$ 346	\$ 152	\$ 788	\$ 322
4	\$ 661	\$ 155	\$ 401	\$ 770
5	\$ 746	\$ 193	\$ 736	\$ 53
6	\$ 762		\$ 614	\$ 200
7	\$ 765		\$ 975	\$ 39
8	\$ 944		\$ 862	\$ 187
9	\$ 52		\$ 92	
10	\$ 768		\$ 535	
11			\$ 214	
12			\$ 501	
13				
14				
15				

FIGURE 11-11

In Figure 11-11, the last used row is 13 because column C contains entries that extend to cell C13. Tomorrow, you might get a similar table, maybe with more columns, where the last used row will be 128 in column K. This is where VBA really shines when you program formulas in R1C1 style when dealing with dynamic ranges. No matter how many columns the table has, or which column has the most entries, the following macro named SumAlongOneRow sums each column's numbers along the first unused row. The result is shown in Figure 11-12.

```
Sub SumAlongOneRow()
'Declare and define a Long type variable for the next available row
'where all the SUM formulas will go.
Dim NextRow As Long
NextRow = _
Cells.Find(What:="*", After:=Range("A1"), _
SearchOrder:=xlByRows, SearchDirection:=xlPrevious).Row + 1
'Declare and define a Long type variable to identify the last column
'in the used range.
Dim LastColumn As Long
LastColumn = _
Cells.Find(What:="*", After:=Range("A1"), _
SearchOrder:=xlByColumns, SearchDirection:=xlPrevious).Column

'The used range starts in column A which is Column 1 in VBA.
'The sales numbers in the table start on row 4.
'Therefore, sum the numbers with a formula that starts at
'row 4 and ends at the last used row, which is one row above
'(numerically 1 less than) the last used row.
Range(Cells(NextRow, 1), Cells(NextRow, LastColumn)).FormulaR1C1 _ 
= "=SUM(R4C:R" & NextRow - 1 & "C)"
End Sub
```

	A	B	C	D
1	We R Widgets, year-to-date sales			
2				
3	Jerry Atrick	Sue Flay	Carrie Okie	Ella Vader
4	\$ 346	\$ 152	\$ 788	\$ 322
5	\$ 661	\$ 155	\$ 401	\$ 770
6	\$ 746	\$ 193	\$ 736	\$ 53
7	\$ 762		\$ 614	\$ 200
8	\$ 765		\$ 975	\$ 39
9	\$ 944		\$ 862	\$ 187
10	\$ 52		\$ 92	
11	\$ 768		\$ 535	
12			\$ 214	
13			\$ 501	
14	\$ 5,044	\$ 500	\$ 5,718	\$ 1,571
15				

FIGURE 11-12

NOTE You probably know that the `RAND` worksheet function enters a random number in a cell. `RAND` is among a group of functions called volatile functions. Volatile functions recalculate whenever another cell in the workbook is changed, or some event takes place such as opening the workbook. You might want to enter a random number and keep it static—that is, for the random number to not change unless you want to change it again, if ever. You can enter a static random number using the following line of code, executable in the Immediate window or as part of a macro. This is an example of how to enter a static random number between 1 and 100 in cell A1. Notice that a value, not actually a formula, is being entered:

```
Range("A1").Value = Format(Rnd() * 99 + 1, "000")
```

TRY IT

For this lesson, you install formulas to sum the numbers in each column of a sales report table. Each column has a varying count of entries, and you want the sum formulas to be placed in the first empty cell below each column's last numeric entry.

Lesson Requirements

To get the sample workbook, you can download Lesson 11 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. In your Excel workbook, press Alt+F11 to go to the Visual Basic Editor.
2. From the VBE menu bar, click Insert \Rightarrow Module.
3. In the new module, type the name of your macro: `SumEachColumnNextRow`. Press Enter, and VBA automatically places a pair of parentheses after the macro name, followed by an empty line, followed by the `End Sub` statement. Your code looks as follows:

```
Sub SumEachColumnNextRow ()  
  
End Sub
```

4. Declare a `Long` type variable to identify the last column in the used range, a `Long` type variable for the last row of numbers present in each column (below which each column's `SUM` formula will go), and a `Long` type variable for the column numbers that will be looped through:

```
Dim LastColumn As Long, lngColumn As Long, LastRow As Long
```

5. Use the `LastColumn` variable inside a loop at each iteration:

```
LastColumn = Cells.Find(What:="*", After:=Range("A1"), _  
SearchOrder:=xlByColumns, SearchDirection:=xlPrevious).Column
```

6. Loop through each column in the used range. The used range starts in column A, which VBA sees as column number 1. Loop through each column and install the formula in the first unused row. While you're at it, bold those sum formula cells to make them easier to see:

```
For lngColumn = 1 To LastColumn  
LastRow = Cells(Rows.Count, lngColumn).End(xlUp).Row  
With Cells(LastRow + 1, lngColumn)  
.FormulaR1C1 = "=SUM(R4C:R" & LastRow & "C)"  
.Font.Bold = True  
End With  
Next lngColumn
```

7. Press Alt+Q to return to the worksheet and test your macro. After you run the macro, the result looks like Figure 11-13. Here's the macro named `SumEachColumnNextRow` in its entirety:

```
Sub SumEachColumnNextRow()  
'Declare a Long type variable to identify the last column  
'in the used range.  
'Declare a Long type variable for the last row of numbers present  
'in each column, below which each column's SUM formula will go.  
'Declare a Long type variable for the column numbers that  
'will be looped through.  
Dim LastColumn As Long, lngColumn As Long, LastRow As Long  
  
'You will use this variable in a loop at each iteration.  
LastColumn = _  
Cells.Find(What:="*", After:=Range("A1"), _  
SearchOrder:=xlByColumns, SearchDirection:=xlPrevious).Column
```

```

'Loop through each column in the used range.
'The used range starts in column A which is Column 1 in VBA.
'Loop through each column and install the formula in the first
'unused row. While we are at it, bold those sum formula cells
'to make them easier to see.
For lngColumn = 1 To LastColumn
    LastRow = Cells(Rows.Count, lngColumn).End(xlUp).Row
    With Cells(LastRow + 1, lngColumn)
        .FormulaR1C1 = "=SUM(R4C:R" & LastRow & "C)"
        .Font.Bold = True
    End With
    Next lngColumn
End Sub

```

	A	B	C	D
1	We R Widgets, year-to-date sales			
2				
3	Jerry Atrick	Sue Flay	Carrie Okie	Ella Vader
4	\$ 346	\$ 152	\$ 788	\$ 322
5	\$ 661	\$ 155	\$ 401	\$ 770
6	\$ 746	\$ 193	\$ 736	\$ 53
7	\$ 762	\$ 500	\$ 614	\$ 200
8	\$ 765		\$ 975	\$ 39
9	\$ 944		\$ 862	\$ 187
10	\$ 52		\$ 92	\$1,571
11	\$ 768		\$ 535	
12	\$5,044		\$ 214	
13			\$ 501	
14			\$5,718	
15				

FIGURE 11-13

REFERENCE Please select the videos for Lesson 11 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

12

Working with Arrays

This lesson introduces you to arrays in VBA. As you will see, arrays are a very useful way to programmatically group and store many items of related data. After you've collected your array of data items, you can access any of the items individually, or access the group as a whole. Arrays can help you accomplish various tasks in a logical and efficient manner, which is important to remember when you find yourself faced with some tasks for which arrays are the only alternative.

WHAT IS AN ARRAY?

An *array* is like a variable on steroids. In addition to being a variable, an array also serves as a holding container for a group of individual values, called *elements*, that are of the same data type. You can populate the array yourself by specifying the known elements in your macro, or you can let VBA populate the array during the course of the macro if you don't know how many elements the array will end up containing.

The concept of arrays can be challenging to grasp at first, so a real-world analogy might help. Suppose you are a fan of classic movies, and you keep a library at home of perhaps 100 movies. Among those 100 movies are 5 that are your favorite classics. You can declare a variable named `myFavoriteMovies`, and create a `String` array with this macro:

```
Sub FavoriteMovies()
    Dim myFavoriteMovies(1 to 5) as String
    myFavoriteMovies (1) = "Gone With The Wind"
    myFavoriteMovies (2) = "Casablanca"
    myFavoriteMovies (3) = "Citizen Kane"
    myFavoriteMovies (4) = "Sunset Boulevard"
    myFavoriteMovies (5) = "Modern Times"
    MsgBox myFavoriteMovies(3)
End Sub
```

Elements in an array are variables, and you can refer to a specific element by its index number inside the array. Because the array name is `myFavoriteMovies`, and the message box is referring to the third element in that array, when you run this macro, the message box displays Citizen Kane.

You have created an array that is a collection of your favorite classic movies. You can loop through each element in that collection—that is, each movie title—by referring to its index number inside the myFavoriteMovies array. The following macro shows how to display each movie title element in a message box:

```
Sub FavoriteMoviesLoop()

Dim myFavoriteMovies(1 To 5) As String
Dim intCounter As Integer

myFavoriteMovies(1) = "Gone With The Wind"
myFavoriteMovies(2) = "Casablanca"
myFavoriteMovies(3) = "Citizen Kane"
myFavoriteMovies(4) = "Sunset Boulevard"
myFavoriteMovies(5) = "Modern Times"

For intCounter = 1 To 5
    MsgBox myFavoriteMovies(intCounter), , "Favorite #" & intCounter
Next intCounter

End Sub
```

If you would like to populate a range of cells with the elements of your array, the following macro demonstrates how to do that, listing the movie titles in range A1:A5:

```
Sub FavoriteMoviesRange()

Dim myFavoriteMovies(1 To 5) As String
Dim intCounter As Integer

myFavoriteMovies(1) = "Gone With The Wind"
myFavoriteMovies(2) = "Casablanca"
myFavoriteMovies(3) = "Citizen Kane"
myFavoriteMovies(4) = "Sunset Boulevard"
myFavoriteMovies(5) = "Modern Times"

For intCounter = 1 To 5
    Cells(intCounter, 1).Value = myFavoriteMovies(intCounter)
Next intCounter

End Sub
```

VBA regards the array itself as one variable, but inside the array is a group of two or more elements that you can work with separately. You can, and often will, refer to each element by its index number, which is its position in the array. This way, you can pick a particular element in the array to work with based on its index number, or you can loop through all the index numbers one after the other, in case your project calls for every element to be worked on.

What Arrays Can Do for You

Arrays are often used for representing data in lists or tables, where each item in the list is of the same data type. Some examples might be a list of your friends' names, all of which would be String data types, or a table of your city's average daily temperatures by month, all of which

might be `Double` data types. Arrays offer you the versatility of storing and manipulating data items through one array variable, which is much more efficient than assigning variables to every element in the array.

Say you want to count how many Excel workbook filenames reside in a particular folder. You don't know how many total files are in that folder, or how many of those total files are Excel files. With an array doing the job, you don't need any worksheet cells to store the filenames. Instead, you can programmatically compile into memory the count of Excel files, and the individual filenames too, all of which you can retrieve later in your macro if need be.

The previous arrays of movie titles are an example of one-dimensional arrays. In the macro named `FavoriteMoviesRange`, the five movies were listed in range A1:A5. VBA regards this as a one-dimensional array because the array elements stand by themselves in a table that is five rows deep and one column wide.

Many arrays you deal with will have more than one dimension. Figure 12-1 expands on this list of classic movies by adding a second column that lists the year each movie was released. This table is composed of five rows and two columns. You can create a two-dimensional `String` array by associating the movie title elements with their respective year of release elements.

	A	B	C
1	Gone With The Wind	1939	
2	Casablanca	1942	
3	Citizen Kane	1941	
4	Sunset Boulevard	1950	
5	Modern Times	1936	
6			

FIGURE 12-1

The first item of business is to declare a `String` type variable for the array. The size of the array is specified with the variable, to include the span of rows and columns that make up the array. For example, with five rows and two columns, a variable named `Classics` is declared with the statement `Dim Classics(1 To 5, 1 To 2) As String`. The following macro loops through rows 1 to 5 in column A and rows 1 to 5 in column B. Each value in the array is stored in memory with two `Integer` type variables for collecting row and column data. Based on Figure 12-1, the message box returns 1941 because `Classics(3, 2)` returns the string value of the element that occupies the location of the array's third row and second column:

```
Sub TwoDimensionalArray()
    Dim Classics(1 To 5, 1 To 2) As String
    Dim intRow As Integer, intColumn As Integer
    For intRow = 1 To 5
        For intColumn = 1 To 2
            Classics(intRow, intColumn) = Cells(intRow, intColumn).Value
        Next intColumn
    Next intRow
    MsgBox Classics(3, 2)
End Sub
```

Declaring Arrays

You declare an array the same way you typically declare variables. The variable declaration starts with the `Dim` statement, followed by the array name and the data type. The array name ends with a pair of parentheses to indicate that it's an array with the count of elements, if known, placed inside the parentheses.

For example, the following statement declares an array named `myDays`, which is populated with all seven days of the week. Notice the data type is `String`, because weekday names are text values, such as “Sunday,” “Monday,” and so on:

```
Dim myDays(6) As String
```

You can also declare arrays using the `Public`, `Private`, and `Static` keywords, just as you can with other variables, with the same results in terms of scope and visibility.

To declare an array as `Public`, place a statement at the top of your module. With the `Public` declaration at the top of your module, you can share an array across procedures. For example, if you run either of the following two macros, the array elements of `Hello` and `Goodbye` will be displayed in a message box:

```
Public MyArray(1) As String

Sub PublicArrayExample()
    'Fill the array MyArray with values.
    MyArray(0) = "Hello"
    MyArray(1) = "Goodbye"
    'Run the TestPublicArrayExample macro to display MyArray.
    Run "TestPublicArrayExample"
End Sub

Sub TestPublicArrayExample()
    'Display the values contained in the array MyArray.
    Dim i As Integer
    For i = 0 To UBound(MyArray, 1)
        MsgBox MyArray(i)
    Next i
End Sub
```

NOTE You may have noticed the `UBound` statement in the preceding macro. You read more about upper and lower boundaries in the upcoming section named “[Boundaries in Arrays](#).”

A `Static` array is an array that is sized in the declaration statement. For example, the following declaration statement declares an `Integer` array that has 11 rows and 11 columns:

```
Dim MyArray(10, 10) as Integer
```

THE OPTION BASE STATEMENT

When learning arrays, it’s common for some head-scratching and confusion to accompany the concept of zero-based numbering. In the declaration statement `Dim myDays(6) As String`, you might wonder why the array shows the number 6 in parentheses, when there are seven days in a week.

In zero-based numbering, the first element of any array is represented by the default number of 0. The second element is represented by the number 1, and so on. That is why an array of seven weekday elements is represented by the number 6 in the statement `Dim myDays(6) As String`.

VBA does provide a way for specifying that the first element of the array be number 1, which is more intuitive for most people. You can do this by placing the statement `Option Base 1` at the top of the module.

NOTE *Most advanced-level VBA programmers exclusively use the default zero-based numbering style. I recommend that you resist the temptation to go the Option Base 1 route in your learning progression. Sooner or later, you will inherit array code that will be zero-based, and you'll be glad you became accustomed to that popular style from the get-go.*

Here's a visual look at zero-based numbering in action.

Figure 12-2 shows five text elements that you might manually place into an array macro.

Note the element index numbers starting with the default of 0. In the following macro, the array named `FamilyArray` is populated in the order of the pictured elements. Further, a variable named `FamilyMember` is assigned the element 2 item, which is actually the third item in the list of names because the list starts at number 0.

Therefore, when the `MsgBox FamilyMember` command is executed, Tom is displayed in the message box because Tom occupies the element 2 position in the array named `FamilyArray`:

```
Sub ArrayTest()
    Dim FamilyArray() As Variant
    Dim FamilyMember As String
    FamilyArray = Array("Bill", "Bob", "Tom", "Mike", "Jim")
    FamilyMember = FamilyArray(2)
    MsgBox FamilyMember
End Sub
```

To test this concept a bit further, enter the statement `Option Base 1` at the very top of the module. When you run the `ArrayTest` macro again, you see that `FamilyArray(2)` returns Bob, because the array elements were counted starting at base number 1.

Element 0	→	Bill
Element 1	→	Bob
Element 2	→	Tom
Element 3	→	Mike
Element 4	→	Jim

FIGURE 12-2

NOTE *It's a fair question to ask why VBA uses zero-based numbering in the first place. Most other programming languages use zero-based numbering for their arrays because of the way arrays are stored in memory. The topic is rather complicated, but in simple English, the subscript (the numbers in the parentheses following the array's variable name) refers to an offset position in memory from the array's starting position. Therefore, the first element has a starting position of 1, but the array's subscript is translated into the offset memory address of 0. The second element is offset at 1, and so on.*

BOUNDARIES IN ARRAYS

Arrays have two boundaries: a lower boundary, which is the position of the first data element, and an upper boundary representing the count of elements in the array. VBA keeps track of both boundaries' values automatically, with the `LBound` and `UBound` functions.

NOTE When you declare an array, you can specify only the upper index boundary. In the example, you have `Dim myDays(6) As String` but it could have been written as `Dim myDays(0 to 6) As String`. The 0 to does not need to be present because the lower index boundary is always assumed to be 0 (or 1 if Option Base 1 has been stated at the top of the module). Under the default setting of Option Base 0, the number you include in the declaration (which was 6 in this example) is the upper index number of the array, not the actual number of elements.

Here is an example to demonstrate the `LBound` and `UBound` functions in practice. In this example, you fill an array with a number of cell addresses, and the macro enters the word Hello in that array of cell ranges:

```
Sub ArraySheets()
    'Declare your variables
    Dim sheetName As Variant, i As Integer, TargetCell as Variant
    'Populate the array yourself with the known cell addresses.
    TargetCell = Array("A1", "B5", "B7", "C1", "C12", "D13", "A12")
    'Loop from the lower boundary (the first array element)
    'to the upper boundary (last element) of your sheetName array.
    For i = LBound(TargetCell) To UBound(TargetCell)
        Range(TargetCell(i)).Value = "Hello"
        'Continue looping through the array elements to completion.
    Next i
    'End the macro.
End Sub
```

DECLARING ARRAYS WITH FIXED ELEMENTS

Early in this lesson you saw this array declaration:

```
Dim myDays(6) As String
```

The ultimate objective of that declaration was to build an array containing the seven days of the week and to transfer that list into range A1:A7, as shown in Figure 12-3.

The macro to do that could look like the following one named `ArrayWeekdays`. Characteristics of a fixed array include a set of elements that remain constant, such as days of the week, where there will always be seven and their names will never change. The `WEEKDAY` function returns an integer from 1 to 7 that represents a day of the week. For example, 1 represents Sunday, 2 represents

	A	B	C
1	Sunday		
2	Monday		
3	Tuesday		
4	Wednesday		
5	Thursday		
6	Friday		
7	Saturday		
8			
9			
10			

FIGURE 12-3

Monday, and so on. If you enter the function =WEEKDAY(5) in a cell, and custom format the cell as DDDD, the cell displays Thursday.

The comments in the code explain what is happening, and why:

```
Sub ArrayWeekdays()
    'Declare the array variable for seven elements (from 0 to 6).
    Dim myDays(6) As String
    'Declare an Integer type variable to handle the seven indexed elements.
    Dim intDay As Integer
    'Start to loop through each array element starting at the default 0 lower boundary.
    For intDay = 0 To 6
        'For each array element, define the myDays String variable
        'with its corresponding day of the week.
        'There is no such thing as "Weekday 0", because Excel's Weekday function
        'is numbered from 1 to 7, so the "+ 1" notation adds 1 to the intDays Integer
        'variable which started at the lower bound of 0.
        myDays(intDay) = Format(Weekday(intDay + 1), "DDDD")
        'Cells in range A1:A7 are populated in turn with the weekday.
        Range("A" & intDay + 1).Value = myDays(intDay)
        'The loop is continued through to conclusion.
    Next intDay
    'End of the macro.
End Sub
```

DECLARING DYNAMIC ARRAYS WITH REDIM AND PRESERVE

Unlike an array with a known fixed set of elements, some arrays are built programmatically during the macro. These arrays are called *dynamic*. Earlier you read about populating an array with the count of Excel workbook files that exist in a folder. In that case you'd have a dynamic array because the file count is subject to change; you would not know ahead of time what the array's size will be. With a dynamic array, you can create an array that is as large or as small as you need to make it.

To attack that problem of an unknown count of elements, you can change the size of an array on the fly with a pair of keywords called `ReDim` and `Preserve`. The `ReDim` statement is short for redimension, a fancy term for resizing the array. When `ReDim` is used by itself to place an element in the array, it releases whatever data was in the array at the time, and simply adds the element to a new empty array.

The `Preserve` statement is necessary to keep (preserve) the data that was in the array, and have the incoming element be added to the existing data. In VBA terms, `ReDim Preserve` raises the array's upper boundary, while keeping the array elements you've accumulated.

The following macro named `SelectedWorksheets` demonstrates `ReDim Preserve` in action. The purpose of the array in this example is to collect the names of all worksheets that are concurrently selected, such as when you press the Ctrl key and select a few worksheet tabs.

The comments in the code explain what each line of code is doing, so you can get a feel for how to populate a dynamic array and display its elements (the worksheet names) in a message box:

```
Sub SelectedWorksheets()
    'Declare the array variable for an unknown count of elements.
    Dim WhatSelected() As Variant
```

```
'Declare a variable for the Worksheet data type.  
Dim wks As Worksheet  
'Declare an Integer variable to handle the unknown count of selected worksheets.  
Dim intSheet As Integer  
'Start to loop through each selected worksheet.  
For Each wks In ActiveWindow.SelectedSheets  
'An index array element is assigned to each selected worksheet.  
intSheet = intSheet + 1  
'This macro is building an array as each selected worksheet is encountered.  
'The Redim statement adds the newest selected worksheet to the growing array.  
'The Preserve statement keeps (preserves) the existing array data,  
'allowing the array to be resized with the addition of the next element.  
ReDim Preserve WhatSelected(intSheet)  
'The corresponding worksheet's tab name is identified with each selected sheet,  
'and placed in the "WhatSelected" array for later retrieval.  
WhatSelected(intSheet) = wks.Name  
'The loop is continued to completion.  
Next wks  
'Looping through each element in the "WhatSelected" array that was just built,  
'a message box displays the name of each corresponding selected worksheet.  
For intSheet = 1 To UBound(WhatSelected)  
    MsgBox WhatSelected(intSheet)  
Next intSheet  
'End of the macro.  
End Sub
```

TRY IT

In this lesson you verify whether a certain string element is part of an array. You test if a certain string element is in an array. At the end of the macro, you show a message box to confirm that the string element either was or was not found to exist in the array.

Like the example earlier in this lesson, say you have this list of names:

- Bill
- Bob
- Tom
- Mike
- Jim

Now, say you want to test whether a certain string element is in that array, which in this example you enter into a worksheet cell. Enter a good-looking name like Tom into cell A1 of Sheet1. Put the list of names in an array, and test to see whether “Tom” is among the elements in that list.

Lesson Requirements

To get the sample workbook you can download Lesson 12 from the book’s website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open Excel and add a new workbook.
2. Press Alt+F11 to get into the Visual Basic Editor.
3. From the VBE menu, select Insert \leftrightarrow Module.
4. In the new module, type the name of your macro:

```
Sub TestArray
```

5. Press the Enter key, which causes Excel to place a set of parentheses after the `TestArray` macro name and also creates the `End Sub` statement. Your macro so far looks like this:

```
Sub TestArray()
```

```
End Sub
```

6. For the first line of code, establish that Sheet2 is `VeryHidden`, as an example to demonstrate the result of an element being found, or not found, in an array. If the element is found, Sheet2 will be unhidden:

```
Worksheets("Sheet2").Visible = xlSheetVeryHidden
```

7. For the second line of code, declare a variable for the array of names you'll be creating, and name the variable `myArray`. For the next line of code, assign the variable name to the array. In this case, you know what the list of names contains so you can build the array yourself by simply entering the individual names inside the parentheses. The two lines of code look like this:

```
Dim myArray As Variant
```

```
myArray = Array("Bill", "Bob", "Tom", "Mike", "Jim")
```

8. The next two lines of code show the `String` type variable to represent the string element you are attempting to verify, and then code to assign the string to that variable. The `String` variable, named `strVerify`, refers to a name you would enter into cell A1 of Sheet1 to test the macro. For example:

```
Dim strVerify as String
```

```
strVerify = Worksheets("Sheet1").Range("A1").Value
```

9. You need to declare two more variables. One of these variables is an `Integer` type variable, which helps you loop through each of the five elements in the array. The other variable is a `Boolean` data type, which helps to characterize as `True` or `False` that the string in cell A1 of Sheet1 is among the elements in the array:

```
Dim i as Integer, blnVerify as Boolean
```

10. Enter `Tom` in cell A1 of Sheet1.

11. Now, to see whether "Tom" exists in the array, loop through each element and compare it to the `String` variable. If there is a match, exit the loop and alert the user by unhiding Sheet2. If the `String` variable is not found, let the user know that as well, and keep Sheet2 hidden:

```
For i = LBound(myArray) To UBound(myArray)
If strVerify = myArray(i) Then
```

```

blnVerify = True
MsgBox "Yes! " & myArray(i) & " is in the array!", , "Verified"
Worksheets("Sheet2").Visible = xlSheetVisible
Exit For
End If
Next i
If blnVerify = False Then _
MsgBox strVerify & " is not in the array.", , "No such animal."

```

12. Putting it all together, the macro looks like this:

```

Sub TestArray ()
'Establish that Sheet2 is VeryHidden.
Worksheets("Sheet2").Visible = xlSheetVeryHidden
'Declare and assign a Variant type variable for the array.
Dim myArray As Variant
myArray = Array("Bill", "Bob", "Tom", "Mike", "Jim")
'Declare and assign a String type variable for the element being evaluated.
Dim strVerify as String
strVerify = Worksheets("Sheet1").Range("A1").Value
'Declare the Integer and Boolean data type variables.
Dim i as Integer, blnVerify as Boolean
'Loop through each element starting with the first one (LBound)
'and continue as necessary through to the last element (UBound).
'If "Tom" is found, exit the loop and alert the user.
'If "Tom" is not found, alert the user of that as well.
For i = LBound(myArray) To UBound(myArray)
If strVerify = myArray(i) Then
blnVerify = True
MsgBox "Yes! " & myArray(i) & " is in the array!", , "Verified"
Worksheets("Sheet2").Visible = xlSheetVisible
Exit For
End If
Next i
If blnVerify = False Then _
MsgBox strVerify & " is not in the array.", , "No such animal."
'End the macro.
End Sub

```

REFERENCE Please select the videos for Lesson 12 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

13

Automating Procedures with Worksheet Events

For the most part, you have run the macros you have seen in this book by pressing a set of shortcut keys, or by going to the Macro dialog box, selecting the macro name, and clicking the Run button. You can take several other actions to run a macro, as you learn in future lessons. The common theme of all these actions is that you have to manually *do something*, whatever it may be, to run a macro.

The question becomes, can a VBA procedure simply know on its own when to run itself, and then just go ahead and do so automatically, without you needing to “do something” to make it run? The answer is yes, and it leads to the subject of event programming, which can greatly enhance the customization and control of your workbooks.

NOTE So far, this book has used the term “macro” to refer to VBA subroutines. When referring to event code, the term “procedure” is used to differentiate it from macro code.

WHAT IS AN EVENT?

In the Excel object model, an *event* is something that happens to an object and is recognized by the computer so an appropriate action can be taken. Recall that the Excel application is made up of objects, such as workbooks, worksheets, cells, charts, pivot tables, and so on. Even the entire Excel application is an object.

Virtually everything you do in Excel is in some way invoking an event upon an object. A few examples of events are as follows:

- Double-clicking a cell
- Adding a worksheet

- Changing a cell value
- Clicking a hyperlink
- Right-clicking a cell
- Calculating a formula

With VBA's event programming capabilities, you can tap into Excel's recognition of when an event occurs and what kind of event it is. This enables you to write VBA code that will execute based on whichever event(s) occur that you want to monitor. This book primarily concentrates on events at two levels:

- *Worksheet-level events*, which are introduced in this lesson.
- *Workbook-level events*, which are introduced in the next lesson.

WORKSHEET EVENTS: AN OVERVIEW

Worksheet-level events occur for a particular worksheet. As you might imagine, events occur when something happens to a worksheet, such as entry of new data into a cell, or a formula being calculated, or the worksheet being activated or deactivated. Event code that is associated with any particular worksheet has no direct effect on events that take place on other worksheets in that or any other workbook.

Where Does the Worksheet Event Code Go?

You've become familiar with the concept of modules as being containers for the macros that you or the Macro Recorder create. You'll be pleased to know that each worksheet already comes with its own built-in module, so you never need to create a module for any worksheet- or workbook-level procedure code.

Worksheet event code always goes into the module of the worksheet for which you are monitoring the event(s). Regardless of the Excel version you are using, the quickest and easiest way to go straight to a worksheet's module is to right-click its sheet tab and select View Code, as shown in Figure 13-1.

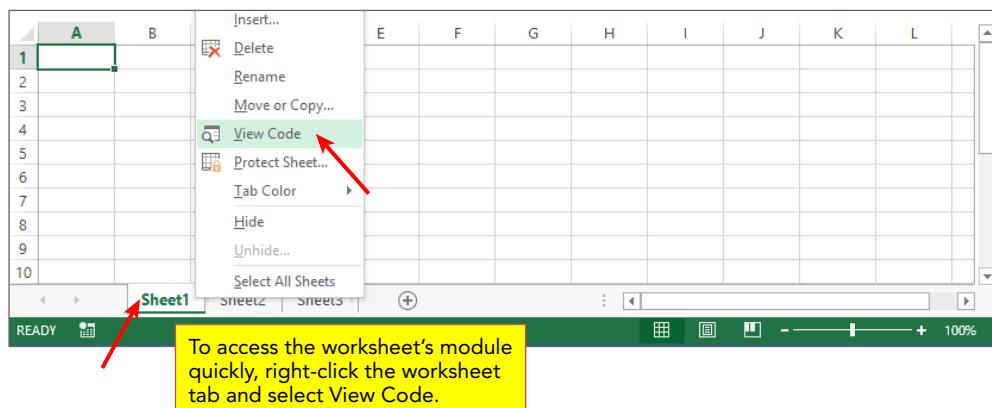


FIGURE 13-1

Immediately after you select View Code, you are taken directly into the Visual Basic Editor, as shown in Figure 13-2. Your mouse cursor will be blinking in the worksheet module's Code window, ready for you to start entering your event procedure code.

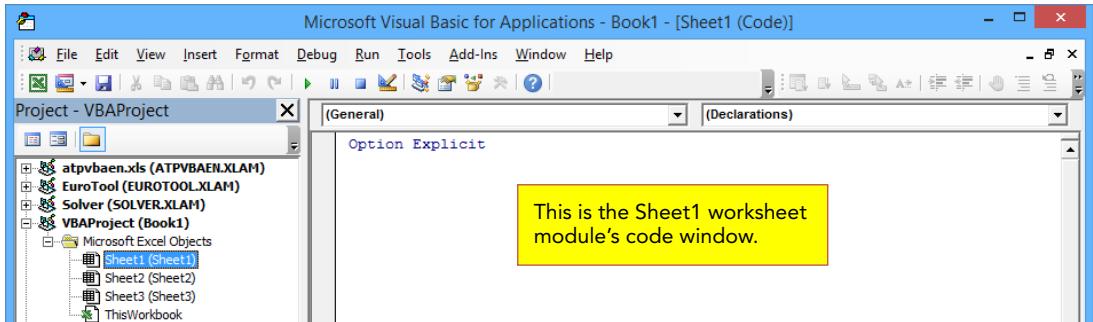


FIGURE 13-2

Immediately above the Code window are two fields with drop-down arrows. The field on the left is the Object field, and when you click its drop-down arrow, you select the Worksheet object item, as shown in Figure 13-3.

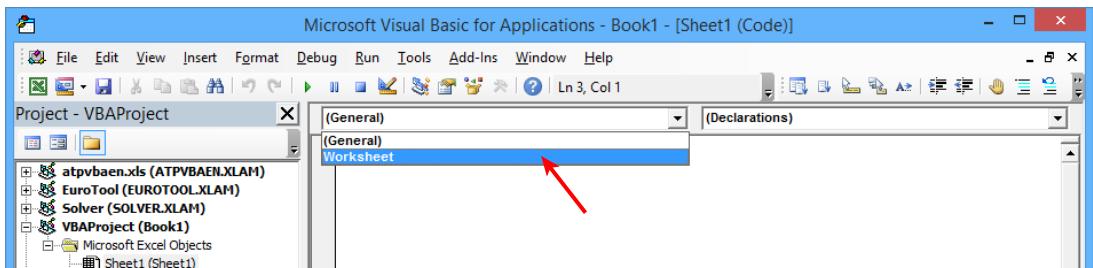


FIGURE 13-3

The field above the worksheet module's Code window, and to the right of the Object field, is the Procedure field. Click the Procedure field's drop-down arrow for a list of the worksheet-level events available to you, as shown in Figure 13-4.

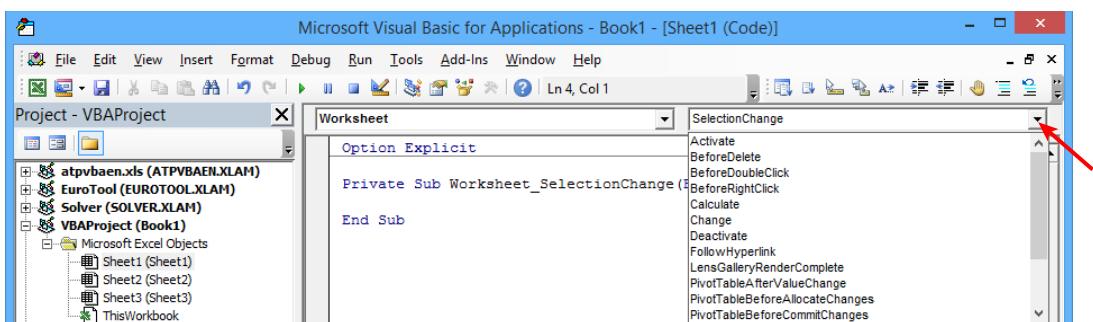


FIGURE 13-4

NOTE When you select an event from the Procedure field's drop-down list, VBA performs the valuable service of entering the procedure statement, with all its argument parameters and an associated End Sub statement, right there in the worksheet module for you.

Enabling and Disabling Events

The Excel Application object has an EnableEvents property that is enabled by default. In some cases you will need to temporarily disable events in your event procedure code, and then re-enable them before the end of the procedure. This may sound strange at first, but the reason is that some events can trigger themselves, and an infinite loop can occur if that happens.

For example, if you are monitoring data entry in a cell and you only want a number to be entered, but a non-numeric entry is attempted, you would use the Worksheet_Change event to undo that wrong entry by clearing the cell's contents. However, VBA regards a cell's contents being cleared as a change having occurred to that cell, which would trigger another round of the same Worksheet_Change event procedure that was already running. To avoid this, you sandwich the relevant code in between statements that disable and enable events, as shown in the following syntax example:

```
Application.EnableEvents = False
'your relevant code
Application.EnableEvents = True
```

NOTE Check out the Try It section at the end of this lesson; it includes two specific examples of disabling and enabling events!

NOTE In the preceding syntax example, the EnableEvents property of the Application object was temporarily set to False with the statement

Application.EnableEvents = False
and then set back to True at the end of the macro with the statement
Application.EnableEvents = True

Keep in mind that the Application object covers all of Excel. For example, while a macro is running with the EnableEvents property of the Application object set to False, EnableEvents is disabled for all open workbooks in that instance of Excel, not just for the workbook where the VBA code is being executed. Whatever properties of the Application object you temporarily change, remember to reset those properties to their original settings before you exit your macro or procedure.

EXAMPLES OF COMMON WORKSHEET EVENTS

At the worksheet level, Excel version 2003 has 9 events, and 5 more than that (associated with pivot tables) for a total of 14 in versions 2007 and 2010. Version 2013 has 3 more events still, for a total of 17.

The additional event procedures in newer versions might be useful for you to learn down the road, but they involve a wider and more specialized instruction of VBA development than the intended introductory scope of VBA in this book. The most commonly used worksheet events are the following nine that are common to all versions of Excel from 2000 to 2013:

- Worksheet_Change
- Worksheet_SelectionChange
- Worksheet_BeforeDoubleClick
- Worksheet_BeforeRightClick
- Worksheet_FollowHyperlink
- Worksheet_Activate
- Worksheet_Deactivate
- Worksheet_Calculate
- Worksheet_PivotTableUpdate

Worksheet_Change Event

The Worksheet_Change event occurs when cells on the worksheet are changed by the user or by an external link, such as a new value being entered into a cell, or the cell's value being deleted. The following example places the current date in column C next to a changed cell in column B:

```
Private Sub Worksheet_Change(ByVal Target As Range)
If Target.Column <> 2 Then Exit Sub
Target.Offset(0, 1).Value = Format(VBA.Date, "MM/DD/YYYY")
End Sub
```

NOTE *The Worksheet_Change event is not triggered by a calculation change, such as a formula returning a different value. Use the Worksheet_Calculate event to capture the changes to values in cells that contain formulas.*

Worksheet_SelectionChange Event

The Worksheet_SelectionChange event occurs when a cell is selected. The following code highlights the active cell with a yellow color every time a different cell is selected:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
Cells.Interior.ColorIndex = 0
Target.Interior.Color = vbYellow
End Sub
```

NOTE A word to the wise! This kind of code is fun and has its uses, but with each change in cell selection, the Undo stack will be eliminated, negating the Undo feature.

Worksheet_BeforeDoubleClick Event

The Worksheet_BeforeDoubleClick event is triggered by double-clicking a worksheet cell. The Cancel argument is optional and halts the ability to go into Edit mode for that cell from a double-click.

In this example, if you double-click a cell in range A1:C8, and the cell already contains a number or is empty, the numeric value of that cell increases by 1. All other cells in the worksheet are unaffected:

```
Private Sub Worksheet_BeforeDoubleClick(ByVal Target As Range, _
Cancel As Boolean)
If Intersect(Target, Range("A1:C8")) Is Nothing Then Exit Sub
If IsNumeric(Target.Value) = True Then
    Cancel = True
    Target.Value = Target.Value + 1
End If
End Sub
```

NOTE This event does not occur if you double-click the active cell's edge, or its fill handle.

Worksheet_Before RightClick Event

The Worksheet_BeforeRightClick event occurs when you right-click a worksheet cell. The optional Cancel argument halts the right-click pop-up menu from appearing. In the following example, when you right-click a cell in column E, the current date and time are entered into that cell and column E's width is autofitted:

```
Private Sub Worksheet_BeforeRightClick(ByVal Target As Range, _
Cancel As Boolean)
If Target.Column <> 5 Then Exit Sub
Cancel = True
Target.Value = Format(VBA.Now, "MMM DD, YYYY, hh:mm AM/PM")
Columns(Target.Column).AutoFit
End Sub
```

Worksheet_FollowHyperlink Event

The Worksheet_FollowHyperlink event occurs when you click any hyperlink on the worksheet. You learn more about command buttons in later lessons, but as a sneak preview, Figure 13-5 shows

a command button embedded onto a worksheet. The button is captioned with a website address but the caption itself is plain text, not actually a hyperlink. With the following code, when you click the command button, you are taken to that caption's website:

```
Private Sub CommandButton1_Click()
    CommandButton1.Parent.Parent.FollowHyperlink CommandButton1.Caption
End Sub
```

	A	B	C	D	E	F	G
1							
2							
3							
4							
5							
6							
7							
8							
9							

<http://www.atlaspm.com>



FIGURE 13-5

NOTE The Worksheet_FollowHyperlink event is available as a worksheet-level event, but in reality, it is more of a function of the entire workbook. Notice the first three items in the statement: CommandButton1.Parent.Parent. The parent of the CommandButton is the worksheet upon which it resides, and the parent of that worksheet is the workbook itself.

A CommandButton as it is referenced in this section is an ActiveX object created from the control toolbox. Another type of button is a Form Button, created from the Form toolbar. ActiveX objects are more complex than Form objects, whereas Form objects are simpler to use and are directly integrated with Excel. ActiveX and Form objects are covered in Lesson 16.

Worksheet_Activate Event

The Worksheet_Activate event occurs when you go to a particular worksheet, typically by clicking the worksheet's tab, although any of the other methods of arriving at a worksheet will trigger the Worksheet_Activate event. Suppose you have a worksheet with one or more pivot tables on it, and every time you go to that worksheet, you want to know that the pivot tables are all refreshed and up to date. The following event code accomplishes that task:

```
Private Sub Worksheet_Activate()
    Dim intCounter As Integer
    For intCounter = 1 To ActiveSheet.PivotTables.Count
        ActiveSheet.PivotTables(intCounter).PivotCache.Refresh
    Next intCounter
End Sub
```

Worksheet_Deactivate Event

The `Worksheet_Deactivate` event occurs when you activate a different worksheet than the one you were on. Suppose there is a particular cell in a worksheet that you strongly prefer to have some value entered into it before the users exit that worksheet. The following `Worksheet_Deactivate` event code checks to see if cell A1 contains a value. If it does not, a message box alerts the users as a reminder of that fact when they deactivate the worksheet:

```
Private Sub Worksheet_Deactivate()
    If Len(Me.Range("A1").Value) = 0 Then
        MsgBox "FYI and reminder: you did not enter a value in cell A1" _
            & vbCrLf & _
            "in the worksheet named " & Me.Name & ".", _
            vbExclamation, _
            "Cell A1 should have some value in it!"
    End Sub
```

Worksheet_Calculate Event

The `Worksheet_Calculate` event occurs when the worksheet is recalculated. Suppose you have a budget model and you want to monitor the bottom-line number for profit and loss, which is derived by a formula in cell Z135. You could conditionally format the cell when its returned value is outside an acceptable range, but chances are no one will see the formatting due to the location of the cell.

To give the budget model's bottom-line Profit/Loss number a boost in awareness, utilize the `Worksheet_Calculate` event to make a message box pop up as a warning when the number in cell Z135 becomes lower than \$1,000. Also, to make it fun, have a congratulatory message appear if the profit number is greater than or equal to \$5,000:

```
Private Sub Worksheet_Calculate()
    If Range("Z135").Value < 1000 Then
        MsgBox "Profits are too low!!", vbExclamation, "Warning!!"
    ElseIf Range("Z135").Value >= 5000 Then
        MsgBox "Profits are TERRIFIC!!", vbExclamation, "Wow, good news!!"
    End If
End Sub
```

Worksheet_PivotTableUpdate Event

The `Worksheet_PivotTableUpdate` event occurs after a pivot table is updated on a worksheet, such as after a refresh. The following procedure is a simple example of the syntax for this event:

```
Private Sub Worksheet_PivotTableUpdate(ByVal Target As PivotTable)
    MsgBox "The pivot table on this worksheet was just updated.", vbInformation, "FYI"
End Sub
```

TRY IT

In this lesson, you write a `Worksheet_Change` event that enables you to sum numbers as they are entered into the same cell. Your `Worksheet_Change` event enables any cell in column A, except for cell A1, to accept a number you enter, add it to whatever number was already in that cell, and

display the resulting sum. For example, if cell A9 currently holds the number 2 and you enter the number 3 in that cell, the resulting value of cell A9 will be 5.

Lesson Requirements

To get the sample workbook you can download Lesson 13 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open a new workbook, right-click the Sheet1 tab, and select View Code.
2. Your cursor will be blinking in the Sheet1 worksheet module. Directly above that, click the down arrow belonging to the Object list, and select Worksheet. This produces the following default lines of code in your worksheet module:

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

```
End Sub
```

3. It is really the Change event you are interested in composing, so take one of two actions: either manually edit the `Private Sub Worksheet_SelectionChange (ByVal Target As Range)` statement by deleting the word Selection, or click the down arrow above the module for the Procedures list, select the Change item, and delete the default `Private Sub Worksheet_SelectionChange (ByVal Target As Range)` statement and its accompanying `End Sub` statement. At this point, the only procedure code you see in your worksheet module is this:

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

```
End Sub
```

4. The event code monitors column A but you want the ability to enter some kind of header label into cell A1. Begin the procedure by writing a line of code to exclude cell A1 from the Change event:

```
If Target.Address = "$A$1" Then Exit Sub
```

5. Your next consideration is to limit the Change event to column A, to avoid imposing the Change event onto the entire worksheet. Also, you want the Change event to be in effect for only one cell at a time in column A. One statement can handle both considerations:

```
If Target.Column <> 1 Or Target.Cells.Count > 1 Then Exit Sub
```

NOTE Note that column A is the first (leftmost) column on the worksheet grid and is easily referred to in VBA as `Columns(1)`. If you had been working with column H, which is the eighth column from the left on the worksheet grid, you would have written this step's line of code as

```
If Target.Column <> 8 Or Target.Cells.Count > 1 Then Exit Sub
```

6. Pressing the Delete key triggers the Change event. You might want to delete a cell's contents and start entering a new set of numbers in an empty cell, so allow yourself the luxury of exiting the Change event if the Delete key is pressed:

```
If IsEmpty(Target) Then Exit Sub
```

7. Even though a number is supposed to be entered into column A, never assume that it will always happen that way, because people make mistakes. Provide for the attempt at a non-numeric entry and disallow it:

```
If IsNumeric(Target.Value) = False Then
```

8. Disable events because you are about to undo the non-numeric value; the Undo command also triggers the Change event:

```
Application.EnableEvents = False
```

9. Execute the Undo action so the non-numeric entry is deleted:

```
Application.Undo
```

10. Enable events again:

```
Application.EnableEvents = True
```

11. Remind the user with a message box that only numbers are allowed, and exit the Change event procedure with the Exit Sub statement:

```
MsgBox "You entered a non-numeric value.", _  
vbExclamation, _  
"Please: numbers only in column A!"  
Exit Sub  
End If
```

12. Now that all the reasonable safeguards have been met, declare two Double type variables: one named OldVal for the numeric value that was in the cell before it was changed, and the other named NewVal for the numeric value that was just entered that triggered this Change event:

```
Dim OldVal As Double, NewVal As Double
```

13. Define the NewVal variable first because it is the number that was just entered into the cell:

```
NewVal = Target.Value
```

14. Undo the entry to display the old (preceding) value. Again, this requires that you disable events so you do not re-trigger the Change event while you are already in a Change event:

```
Application.EnableEvents = False
```

15. Execute Undo so the previous value is re-established:

```
Application.Undo
```

16. Define the OldVal variable, which is possible to do now that the previous value has been restored:

```
OldVal = Target.Value
```

17. Programmatically enter into the cell the sum of the previous value, plus the new last-entered value, by referring to those two variables in an arithmetic equation just as you would if they were numbers:

```
Target.Value = OldVal + NewVal
```

18. Enable events now that all the changes to the cell have been made:

```
Application.EnableEvents = True
```

19. When completed, the entire procedure looks like this, with comments that have been added to explain each step:

```
Private Sub Worksheet_Change(ByVal Target As Range)
    'Allow for a header label to be placed in cell A1.
    If Target.Address = "$A$1" Then Exit Sub
    'Only apply this effect to column A (column 1 in VBA-Speak).
    'At the same time, only allow one cell at a time to be changed.
    If Target.Column <> 1 Or Target.Cells.Count > 1 Then Exit Sub
    'Pressing the Delete key triggers the Change event.
    'You might want to delete the cell's contents and start with
    'an empty cell, so exit the Change event if the Delete key is pressed.
    If IsEmpty(Target) Then Exit Sub

    'Even though a number is *supposed* to be entered into column A,
    'never assume that will always happen because users do make mistakes.
    'Provide for the attempt at a non-numeric entry and disallow it.
    If IsNumeric(Target.Value) = False Then
        'Disable events because you are about to undo the non-numeric value,
        'and Undo also triggers the Change event.
        Application.EnableEvents = False
        'Execute the Undo so the non-numeric entry is deleted.
        Application.Undo
        'Enable events again.
        Application.EnableEvents = True
        'Remind the user with a Message Box that only numbers are allowed,
        'and exit the Change event procedure with the Exit Sub statement.
        MsgBox "You entered a non-numeric value.", _
            vbExclamation, _
            "Please: numbers only in column A!"
        Exit Sub
    End If

    'Now that all the reasonable safeguards have been met,
    'Declare two Double type variables:
    'one named OldVal for the numeric value that was in the cell
    'before it got changed,
    'and the other variable named NewVal for the numeric value
    'that was just entered that triggered this Change event.
    Dim OldVal As Double, NewVal As Double
    'Define the NewVal variable first, as it is the number that
    'was just entered into the cell.
```

```
NewVal = Target.Value
'Undo the entry in order to display the old (preceding) value.
'Again, this requires that you disable events in order to not
're-trigger the Change event while you are already in a Change event.
Application.EnableEvents = False
'Execute Undo so the previous value is re-established.
Application.Undo
'Define the OldVal variable which is possible to do now that
'the previous value has been restored.
OldVal = Target.Value
'Programmatically enter into the cell the sum of the old previous value,
'plus the new last-entered value, by referring to those two variables
'in an arithmetic equation just as you would if they were numbers.
Target.Value = OldVal + NewVal
'Enable events now that all the changes to the cell have been made.
Application.EnableEvents = True
End Sub
```

20. Press Alt+Q to return to the worksheet. Test the code by entering a series of numbers in any single cell in column A other than cell A1.

REFERENCE Please select the video for Lesson 13 at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

14

Automating Procedures with Workbook Events

In Lesson 13, you learned about worksheet-level events and how they are triggered by actions relating to individual worksheets. Workbooks themselves can also recognize and respond to a number of events that take place at the workbook level. This lesson describes how you can further customize your workbooks with VBA procedures for the most commonly used workbook events.

WORKBOOK EVENTS: AN OVERVIEW

Workbook events occur within a particular workbook. Many workbook events occur because something happened to an object in the workbook, such as a worksheet—any worksheet—that was activated, or a cell—any cell—that was changed. Other workbook events occur because the workbook was imposed upon to do something, such as to open or close, or to be saved or printed.

NOTE *Unless the VBA code itself purposely refers to other workbooks, event procedures at the workbook level affect only the workbook within which the code resides.*

Where Does the Workbook Event Code Go?

You saw in Lesson 13 that each individual worksheet has its own module. Workbooks are similar to worksheets in this respect because a workbook is also an Excel object, and it has its own module already present and accounted for when the workbook is created.

NOTE Workbook-level event code always goes into the workbook module. You never need to create a workbook module or a worksheet module; Excel creates those modules automatically with every new workbook. If a workbook-level event procedure is not in the workbook module (same as if a worksheet-level event procedure is not in a worksheet module), VBA will not be able to execute the event code.

To arrive at the Code window for your workbook's module, with whatever version of Excel you are using, you can press Alt+F11 to get into the Visual Basic Editor. If you are using a version of Excel prior to 2007, such as version 2003, you can also access the workbook module quickly by right-clicking the Excel workbook icon near the top-left corner of the workbook window and selecting View Code. This option is shown in Figure 14-1.

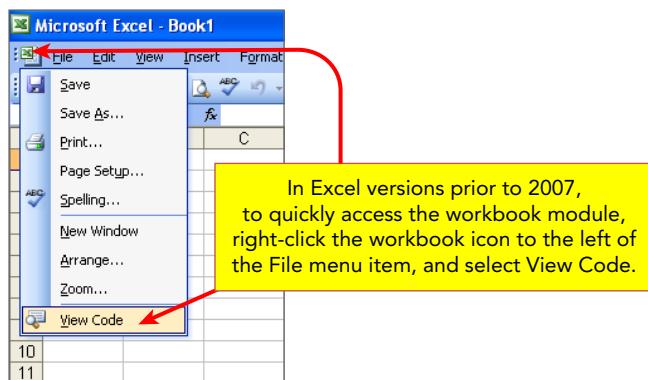


FIGURE 14-1

In the VBE, if you do not see the Project Explorer window, go ahead and make it visible by pressing Ctrl+R. In the Project Explorer, find your workbook name; it is in bold font, with the text `VBAProject (YourWorkbookName.xlsxm)`. Directly below that will be a yellow folder named Microsoft Excel Objects. When you expand that folder, the last item at the bottom of the list is your workbook object, identified by its default name of `ThisWorkbook`.

NOTE You saw in Lesson 4 how to change the name of a module. You can change the name of the workbook module, but do yourself a favor and leave the workbook module's default name alone. The `ThisWorkbook` module name is consistent with 99.99 percent of VBA workbook projects you'll likely encounter. There's almost never a benefit to be gained by changing the workbook module's name.

As shown in Figure 14-2, to get into the Code window of the workbook module, either double-click the `ThisWorkbook` object, or right-click it and select View Code. As soon as you do that, your

mouse cursor will be blinking in the workbook module's Code window, ready for you to start entering your workbook-level event procedure code.

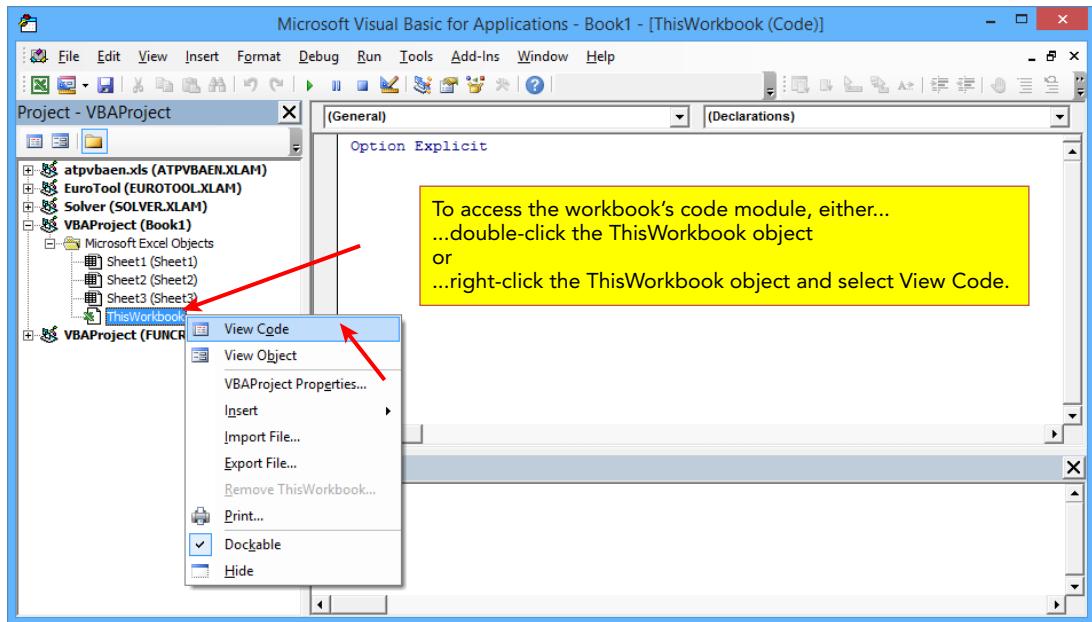


FIGURE 14-2

Entering Workbook Event Code

Similar to the worksheet module Code window you saw in Lesson 13, two fields with drop-down arrows are located above the workbook module's Code window. The field on the left is the Object field, and when you click its drop-down arrow, you select the Workbook object item, as shown in Figure 14-3.

The field above the workbook module's Code window, and to the right of the Object field, is the Procedure field. Click the Procedure field's drop-down arrow for a list of the workbook-level events available to you, as shown in Figure 14-4.

NOTE *For convenience, just as with selecting worksheet-level event names, VBA places the complete workbook-level event statement, with all its arguments and the accompanying End Sub statement, when you select a workbook-level event name from the Procedure field.*

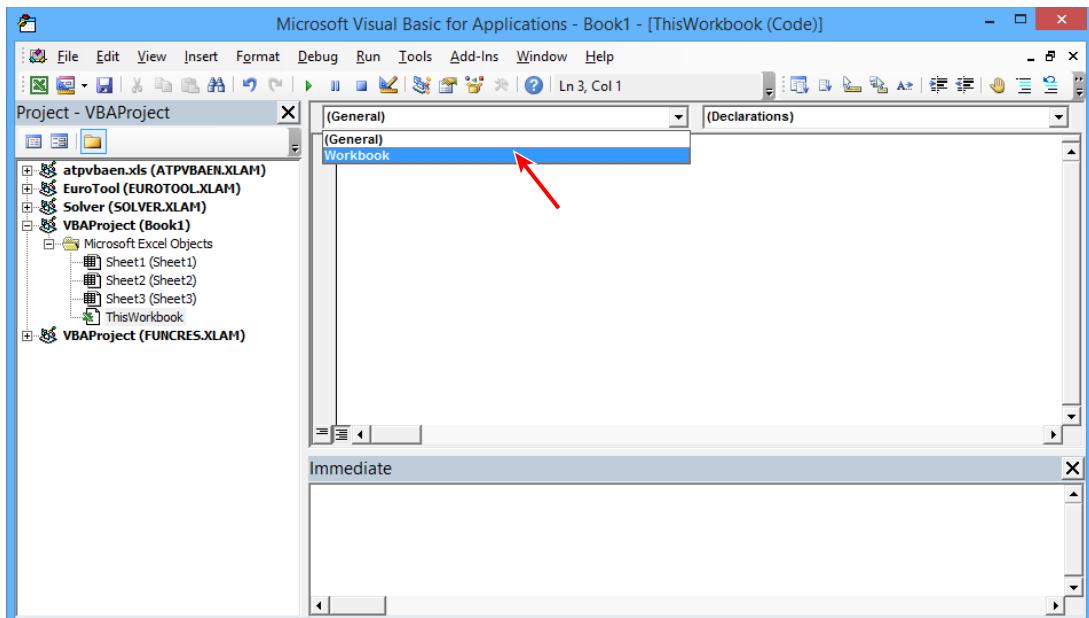


FIGURE 14-3

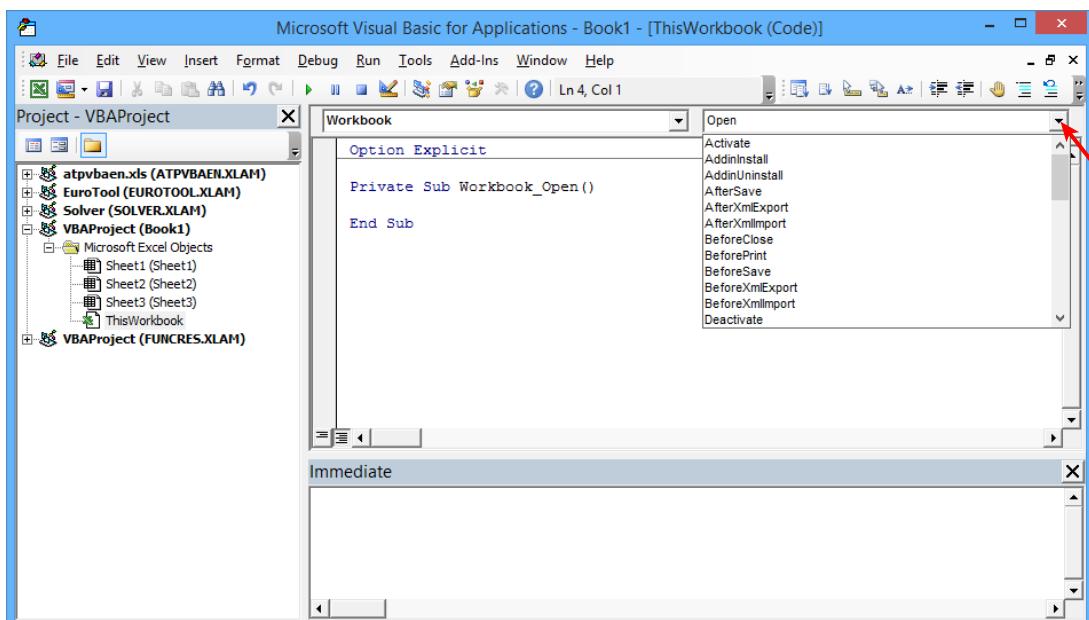


FIGURE 14-4

EXAMPLES OF COMMON WORKBOOK EVENTS

At the workbook level, Excel version 2003 has 28 events, and 8 more than that (mostly associated with pivot tables) for a total of 36 in versions 2007 and 2010. Microsoft added 4 more events to version 2013 for a total of 40. The most commonly used workbook-level events across all versions of Excel are listed here, with examples of each on the following pages:

- Workbook_Open
- Workbook_BeforeClose
- Workbook_Activate
- Workbook_Deactivate
- Workbook_SheetChange
- Workbook_SheetSelectionChange
- Workbook_SheetBeforeDoubleClick
- Workbook_SheetBeforeRightClick
- Workbook_SheetPivotTableUpdate
- Workbook_NewSheet
- Workbook_BeforePrint
- Workbook_SheetActivate
- Workbook_SheetDeactivate
- Workbook_BeforeSave

Workbook_Open Event

The Workbook_Open event is triggered when the workbook opens, and is among the most popular and useful of all workbook-level events. The Workbook_Open event is perfect for such tasks as informing users about important features of your workbook, or generating a running list of users who have accessed the workbook, or establishing a particular format setting that would be reset to its original state with the Workbook_BeforeClose event.

In the Try It section of Lesson 13, you saw an example of how to enter a number in a cell and have that number be added to whatever number was previously in that cell. The users of this workbook might appreciate knowing about, or being reminded of, that capability when they open the workbook. You can use the Workbook_Open event, such as with the following procedure, for example, to show a message box that informs the users of that workbook's special capability:

```
Private Sub Workbook_Open()
MsgBox _
"FYI, when you enter a number in a cell in column A" & vbCrLf & _
"of Sheet3, it will automatically be added to the" & vbCrLf & _
"number previously in that cell, and display the sum.", _
vbInformation, _
```

```
"Welcome! Here's a tip for this workbook:  
End Sub
```

Workbook_BeforeClose Event

The Workbook_BeforeClose event is triggered just before the workbook closes. This event is often used in conjunction with the Workbook_Open event, to set a workbook back to its original state if the Workbook_Open event temporarily changed the user's Excel settings.

The following example is one way to apply the Workbook_BeforeClose event's usefulness. You can tell Excel to save your workbook automatically when you close it, to avoid Excel's prompt that asks you if you want to save your changes (and losing your work if you mistakenly were to click No!):

```
Private Sub Workbook_BeforeClose(Cancel As Boolean)  
ThisWorkbook.Save  
End Sub
```

Workbook_Activate Event

The Workbook_Activate event is triggered when the workbook is activated, such as when the workbook is opened, or when you switch between that workbook and other open workbooks. In this example, the following procedure maximizes the Excel window when you activate the workbook:

```
Private Sub Workbook_Activate()  
ActiveWindow.WindowState = xlMaximized  
End Sub
```

Workbook_Deactivate Event

The Workbook_Deactivate event is triggered when the workbook loses focus, such as when a different Excel workbook is activated or when the workbook is closed. The following example prompts a message box to alert you when the workbook is deactivated:

```
Private Sub Workbook_Deactivate()  
MsgBox "You are leaving " & Me.Name & "!", _  
vbInformation,  
"Just saying..."  
End Sub
```

Workbook_SheetChange Event

The Workbook_SheetChange event is triggered when any cell's contents are changed on any worksheet in the workbook. If you would like to keep a log of the date, time, sheet name, and address of any cell that gets changed, this procedure accomplishes that by listing information on a worksheet named Log:

```
Private Sub Workbook_SheetChange(ByVal Sh As Object, ByVal Target As Range)  
'The log sheet will hold the record of each sheet change,  
'so halt the event if a cell is changed on the Log sheet.  
If Sh.Name = "Log" Then Exit Sub  
'Declare a Long variable for the next available row on the Log sheet.  
Dim NextRow As Long
```

```

'Assign the row number to the next empty row below that last row of data
'in column A.
NextRow = Worksheets("Log").Cells(Rows.Count, 1).End(xlUp).Row + 1
'In column A, enter the date of the changed cell.
Worksheets("Log").Cells(NextRow, 1).Value = VBA.Date
'In column B, enter the time of the changed cell.
Worksheets("Log").Cells(NextRow, 2).Value = VBA.Time
'In column C, enter the name of the worksheet holding the changed cell.
Worksheets("Log").Cells(NextRow, 3).Value = Sh.Name
'In column D, enter the address of the changed cell.
Worksheets("Log").Cells(NextRow, 4).Value = Target.Address
'Autofit the columns on the Log sheet, to make the information readable.
Worksheets("Log").Columns.AutoFit
End Sub

```

Workbook_SheetSelectionChange Event

The `Workbook_SheetSelectionChange` event is triggered when a different cell is selected on any worksheet in the workbook. In Lesson 13, you saw an example of the `Worksheet_SelectionChange` event whereby the active cell was continuously highlighted. If you are navigating through large ranges of data on your worksheets, such as budgets or financial reports, you might find it useful to visually identify more than just the active cell. The following procedure highlights the entire row and column at each new cell selection:

```

Private Sub Workbook_SheetSelectionChange(ByVal Sh As Object, _
 ByVal Target As Range)
Dim myRow As Long, myColumn As Long
myRow = Target.Row
myColumn = Target.Column
Sh.Cells.Interior.ColorIndex = 0
Sh.Rows(myRow).Interior.Color = vbGreen
Sh.Columns(myColumn).Interior.Color = vbGreen
End Sub

```

Workbook_SheetBeforeDoubleClick Event

The `Workbook_SheetBeforeDoubleClick` event is triggered when a cell on any worksheet is about to be double-clicked. The double-click effect (usually getting into Edit mode) can be canceled with the `Cancel` parameter.

Suppose you have a workbook wherein column A of every worksheet is reserved for the purpose of placing check marks in cells. You do not want to deal with embedding possibly hundreds of real check box objects, so a check mark-looking character in a cell would suffice.

You can utilize the `Workbook_DoubleClick` event that would apply only to column A for any worksheet. The following procedure toggles the effect of placing a check mark in column A. If the cell is empty, a check mark is entered, and if a check mark is present when the cell is double-clicked again, the check mark is removed. As you can see in the code, the “check mark” is really a lowercase letter “a” formatted in Marlett font:

```

Private Sub Workbook_SheetBeforeDoubleClick(ByVal Sh As Object, _
 ByVal Target As Range, Cancel As Boolean)

```

```
If Target.Column <> 1 Then Exit Sub
Cancel = True
Target.Font.Name = "Marlett"
Target.HorizontalAlignment = xlCenter
If IsEmpty(Target) = True Then
    Target.Value = "a"
Else
    Target.Clear
End If
End Sub
```

Workbook_SheetBeforeRightClick Event

The `Workbook_SheetBeforeRightClick` event is triggered when a cell on any worksheet is about to be right-clicked. You can cancel the right-click effect of the pop-up menu with the `Cancel` parameter.

Suppose you want to add a utility to your workbook that would enable you to quickly and easily insert a row above any cell you right-click. A message box could ask if you want to insert a row, and if you answer yes, a row would be inserted. The following procedure is an example of how you can handle that:

```
Private Sub Workbook_SheetBeforeRightClick(ByVal Sh As Object, _
    ByVal Target As Range, Cancel As Boolean)
    If MsgBox("Do you want to insert a row here?", _
        vbQuestion + vbYesNo, _
        "Please confirm...") = vbYes Then
        Cancel = True
        ActiveCell.EntireRow.Insert
    End If
End Sub
```

Workbook_SheetPivotTableUpdate Event

The `SheetPivotTableUpdate` event monitors all worksheets in the workbook that hold pivot tables. In the following event code, when a pivot table is updated, the name of its worksheet appears in a message box.

```
Private Sub Workbook_SheetPivotTableUpdate(ByVal Sh As Object, _
    ByVal Target As PivotTable)
    MsgBox "The pivot table on sheet " & Sh.Name & " was updated.", , "FYI"
End Sub
```

Workbook_NewSheet Event

The `Workbook_NewSheet` event is triggered when a new sheet is added to the workbook. To see this event in action, suppose you do not want to formally protect the workbook, but you want to disallow the addition of any new worksheets. This event procedure promptly deletes a new sheet as soon as it is added, with a message box informing the user that adding new sheets is not permitted:

```
Private Sub Workbook_NewSheet(ByVal Sh As Object)
    Dim asn As String
    asn = ActiveSheet.Name
```

```

Application.EnableEvents = False
Application.DisplayAlerts = False
Sheets(ActiveSheet.Name).Delete
MsgBox "Sorry, new sheets are not allowed to be added.", vbCritical, " FYI"
Application.DisplayAlerts = True
Application.EnableEvents = True
End Sub

```

Workbook_BeforePrint Event

The `Workbook_BeforePrint` event is triggered before a user attempts to print any portion of the workbook. You can cancel the print job by setting the `Cancel` parameter to `True`. If you want to ensure that anything printed from that workbook will have the workbook's full name in the footer of every printed page, the following procedure accomplishes that:

```

Private Sub Workbook_BeforePrint(Cancel As Boolean)
Dim sht As Worksheet
For Each sht In ThisWorkbook.Sheets
    sht.PageSetup.CenterFooter = ThisWorkbook.FullName
Next sht
End Sub

```

Workbook_SheetActivate Event

The `Workbook_SheetActivate` event is triggered when a sheet is activated in the workbook. Suppose you want to always return to cell A1 whenever you activate any worksheet, regardless of what cell you had selected the last time you were in that worksheet. The following procedure using the `Application.GoTo` statement does just that:

```

Private Sub Workbook_SheetActivate(ByVal Sh As Object)
If TypeName(Sh) = "Worksheet" Then Application.Goto Range("A1"), True
End Sub

```

NOTE *This example illustrates the distinction between a Sheet object and a Worksheet object—they are not necessarily the same things. Excel has several types of Sheet objects: worksheets, chart sheets, outdated dialog sheets, and the obsolete macro sheets. In this example, a chart sheet would create confusion for VBA because chart sheets do not contain cells. Only worksheets contain cells, which is why the TypeName of Worksheet is the only Sheet object at which this procedure's code is directed.*

Workbook_SheetDeactivate Event

The `Workbook_SheetDeactivate` event is triggered when a sheet loses focus, such as when a different sheet in the workbook is activated. If you have a workbook with tables of data on every worksheet, and you want the tables to be sorted automatically by column A whenever you leave the worksheet, this procedure does that:

```
Private Sub Workbook_SheetDeActivate(ByVal Sh As Object)
If TypeName(Sh) = "Worksheet" Then
    Sh.Range("A1").CurrentRegion.Sort Key1:=Sh.Range("A2"), _
    Order1:=xlAscending, Header:=xlYes
End If
End Sub
```

Workbook_BeforeSave Event

The Workbook_BeforeSave event is triggered just before the workbook is saved. You can set the Cancel parameter to True to stop the workbook from being saved.

Suppose you want to limit the time period for a workbook to be saved. The following procedure allows the workbook to be saved only between 9:00 AM and 5:00 PM:

```
Private Sub Workbook_BeforeSave(ByVal SaveAsUI As Boolean, Cancel As Boolean)
If VBA.Time < TimeValue("09:00") Or VBA.Time > TimeValue("17:00") Then Cancel = True
End Sub
```

TRY IT

In this lesson you write a Workbook_BeforePrint workbook-level event that instructs Excel not to print a particular range of confidential data that resides on a particular worksheet.

Lesson Requirements

To get the sample database files you can download Lesson 14 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open a new workbook and activate Sheet3. To prepare the worksheet for this demonstration, populate range A1:E20 with some sample data by selecting the range, typing the word **Hello**, and pressing Ctrl+Enter.
2. On your keyboard, press Alt+F11 to go to the Visual Basic Editor, and then press Ctrl+R to ensure that the Project Explorer window is visible.
3. Find the name of your workbook in the Project Explorer, and expand the folder named Microsoft Excel Objects.
4. The last item at the bottom of the list of Microsoft Excel Objects is the workbook object, and it is called **ThisWorkbook**. You'll want to access the Code window for the **ThisWorkbook** module, and to do that, you can either double-click the **ThisWorkbook** object name or right-click it and select View Code.
5. The cursor will be blinking in the Code window of your workbook module. Directly above that, click the down arrow belonging to the Object list, and select Workbook, which produces the following default lines of code in your workbook module:

```
Private Sub Workbook_Open()
End Sub
```

6. In this example you write a `BeforePrint` procedure, so click the other down arrow above the Code window for the Procedure field, and select `BeforePrint`. VBA produces these lines of code, which is just what you want:

```
Private Sub Workbook_BeforePrint(Cancel As Boolean)
End Sub
```

7. Though not imperative, unless you are planning to employ the `Workbook_Open` event, there's no reason to keep the default `Private Sub Workbook_Open()` and `End Sub` statements, so go ahead and delete them if you like.

8. In this example, you have confidential data on Sheet3 only, so instruct Excel that it's okay to print anything on any worksheet other than Sheet3:

```
If ActiveSheet.Name <> "Sheet3" Then Exit Sub
```

9. Invoke the `Cancel` argument to halt the print process when an attempt is made to print Sheet3:
`Cancel = True`

10. Disable events because you actually will be printing something, but you don't want to re-trigger the `BeforePrint` event while you are already in it:

```
Application.EnableEvents = False
```

11. Your confidential data resides in range B5:D12. Temporarily format that range with three semicolons to make those cells unable to display their contents:

```
Range("B5:D12").NumberFormat = ";;;"
```

12. Print the worksheet:

```
ActiveSheet.PrintOut
```

NOTE When you test the `Workbook_BeforePrint` procedure, you can use the `PrintPreview` method instead of the `PrintOut` method, which can save you costs in paper and printer toner.

13. Restore the General format to the confidential range so the cells will be able to show their contents after the print job:

```
Range("B5:D12").NumberFormat = "General"
```

14. Enable events again, now that the print job has been executed:

```
Application.EnableEvents = True
```

15. When completed, the entire procedure looks like this, with comments that have been added to explain each step:

```
Private Sub Workbook_BeforePrint(Cancel As Boolean)
```

```
'You have confidential data on Sheet3 only,
'so any other sheet is OK to print anything.
If ActiveSheet.Name <> "Sheet3" Then Exit Sub
'Invoke the Cancel argument to halt the print process.
Cancel = True
'Disable events because you actually will print something
'but you don't want the BeforePrint event to kick in.
Application.EnableEvents = False
'Your confidential data resides in range B5:D12.
'Temporarily format that range with three semicolons
'to make those cells unable to display their contents.
Range("B5:D12").NumberFormat = ";;;"
'Print the worksheet.
ActiveSheet.PrintOut 'demo with PrintPreview
'Restore the General format to the confidential range
'so the cells will be able to show their contents
'after the print job.
Range("B5:D12").NumberFormat = "General"
'Enable events again, now that the print job has been executed.
Application.EnableEvents = True
End Sub
```

16. Press Alt+Q to return to the worksheet. Test the code by printing Sheet3, noting that the printout shows an empty range of cells, representing the range of confidential data that did not get printed.

REFERENCE Please select the video for Lesson 14 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

15

Handling Duplicate Items and Records

When you work with data in tables or lists, it is common for some items to appear more than once. Two situations usually arise when duplicate items exist, depending on the nature of the work at hand:

- The repeated items are unwanted and need to be deleted. For example, if you are compiling a list of e-mail addresses, or you are gathering a list of people's names for invitation to an event, you would only want a list of unique items.
- Items are expected to be repeated in the list and need to be maintained for analysis or record-keeping. For example, a list of monthly payments made to a vendor would show that vendor's name with each transaction.

DELETING ROWS CONTAINING DUPLICATE ENTRIES

Suppose a table of data contains duplicate items in one or more columns. To delete rows containing duplicate items, the first step is to determine if the table contains duplicates in just one column, or if several (maybe all) columns contain duplicate data.

Deleting Rows with Duplicates in a Single Column

Suppose you have a list of items that are repeated in column A. The macro named `DeleteDuplicatesColumnA` uses `AdvancedFilter` to expose the first instance of every item in column A. The exposed rows are marked with a value (the numeral 1 in this example, but it could be any value) in a helper column. All rows with empty cells in the helper column are deleted.

NOTE *For my money, AdvancedFilter is the second-most powerful tool in Excel, behind pivot tables. One of AdvancedFilter's capabilities is to filter for unique items in a large list at lightning speed.*

The macro executes in the blink of an eye, even for a list with tens of thousands of rows. There are comments at each step to explain the deletion process using AdvancedFilter:

```
Sub DeleteDuplicatesColumnA()
    'Long variable for last used column.
    Dim LastColumn As Long
    With Application
        .ScreenUpdating = False
        'Determine last column number of table, and add a 1 to it
        'to establish a helper column that is the column after
        'the last column in the data table.
        LastColumn =
            Cells.Find(What:="*", After:=Range("A1"), SearchOrder:=xlByColumns, _
            SearchDirection:=xlPrevious).Column + 1
        'AdvancedFilter exposes unique entries and enters a 1 in the helper
        'column on the same row of items that first appear in the list.
        With Range("A1:A" & Cells(Rows.Count, 1).End(xlUp).Row)
            .AdvancedFilter Action:=xlFilterInPlace, Unique:=True
            .SpecialCells(xlCellTypeVisible).Offset(0, LastColumn - 1).Value = 1
            'Error bypass that is explained in Lesson 20.
            'This is to avoid the macro stopping if no duplicate values existed.
        On Error Resume Next
            'Show all rows by exiting AdvancedFilter.
        ActiveSheet.ShowAllData
            'Delete rows where empty cells exist in the helper column,
            'indicating that the value in column A is a duplicate.
            Columns(LastColumn).SpecialCells(xlCellTypeBlanks).EntireRow.Delete
        Err.Clear
    End With
    'Clear the helper column.
    Columns(LastColumn).Clear
    .ScreenUpdating = True
End With
End Sub
```

Your lists will not always have its duplicate entries in column A. The next list you receive might have its duplicate entries in a different column, say column D. The DeleteDuplicatesColumnD macro is a modification of the previous macro, with comments showing where and how to change the relevant column references:

```
Sub DeleteDuplicatesColumnD()
    'Ask the user to confirm their intention of deleting
    'the duplicate items in column D.
    Dim myConfirmation As Integer
    myConfirmation =
        MsgBox("Do you want to delete the duplicates" & vbCrLf & _
        "in column D?" & vbCrLf & vbCrLf & _
        "Once the duplicates are deleted," & vbCrLf & _
        "the macro cannot undo that action.", _
        vbQuestion + vbYesNo, _
        "Please confirm:")
```

```

'If the answer is no, exit the macro.
If myConfirmation = vbNo Then
    MsgBox "That's fine, nothing will be deleted.", _
        vbInformation, _
        "You clicked No."
    Exit Sub

Else

    'If the answer is yes, continue with the deletion.
    MsgBox "Please click OK to delete the duplicates.", _
        vbInformation, _
        "Thanks for confirming!"

End If

With Application
    .ScreenUpdating = False
    Dim LastColumn As Long
    LastColumn = _
        Cells.Find(What:="*", After:=Range("A1"), SearchOrder:=xlByColumns, _
        SearchDirection:=xlPrevious).Column + 1

    'In the next line you specify column D.
    'If this were for column L instead of column D, the code would read
    'With Range("L1:L" & Cells(Rows.Count, 12).End(xlUp).Row)
    With Range("D1:D" & Cells(Rows.Count, 4).End(xlUp).Row)
        .AdvancedFilter Action:=xlFilterInPlace, Unique:=True
        'In the next line, notice the number 4 in the Cells property,
        'which is column D. If this were for column L instead of column D,
        'number 4 would be 12, example .Offset(0, LastColumn - 12).Value = 1
        .SpecialCells(xlCellTypeVisible).Offset(0, LastColumn - 4).Value = 1

    On Error Resume Next
    ActiveSheet.ShowAllData
    Columns(LastColumn).SpecialCells(xlCellTypeBlanks).EntireRow.Delete
    Err.Clear
    End With
    Columns(LastColumn).Clear
    .ScreenUpdating = True
End With
End Sub

```

NOTE Notice in these macros that no cell or row is selected, which would have slowed things down, and a filter is utilized for the entire range instead of a loop for each row. When deleting rows, use a filter when you can because it is much faster than looping through cells one by one.

Please keep in mind that there is not an undo option after a macro runs. It's a wise practice to let the users of your projects know the consequence of running a macro that deletes data. For example, the DeleteDuplicatesColumnD macro begins with a message box to inform the user that the macro's actions cannot be undone, and to confirm their intention to delete the duplicate items.

Deleting Rows with Duplicates in More Than One Column

When you have a list of data, sometimes it is not enough to simply delete rows with duplicated information based only on the items in one column. Multicolumn lists can have duplicated records when every item in every column of a row's data matches that of another row's entire data. In those cases, you need to compare a concatenated string of each record's (row's) data, and compare that to the concatenated strings of all the other rows.

Take a close look at Figure 15-1. In the original list, every item in rows 5 and 7 match, as do all the items in rows 3 and 10. This is a short list for demonstration purposes. If your list were thousands of rows long, you would need a quick way to delete duplicate records. The macro named DeleteDuplicateRecords is one way to do the job, with comments at each step.

Original list with 2 duplicate records.				
	A	B	C	D
1	Last	First	Street Address	City
2	Smith	John	123 Main Street	Toledo
3	Doe	Jane	123 Mack Street	Xenia
4	Smith	Jane	123 Mean Street	Toledo
5	Jones	John	123 Main Street	Xenia
6	Doe	John	123 Mack Street	Xenia
7	Jones	John	123 Main Street	Xenia
8	Smith	Jane	123 Main Street	Toledo
9	Doe	John	123 Mack Street	Toledo
10	Doe	Jane	123 Mack Street	Xenia
11	Jones	John	123 Mean Street	Toledo

The two duplicate records are deleted.				
	A	B	C	D
1	Last	First	Street Address	City
2	Smith	John	123 Main Street	Toledo
3	Doe	Jane	123 Mack Street	Xenia
4	Smith	Jane	123 Mean Street	Toledo
5	Jones	John	123 Main Street	Xenia
6	Doe	John	123 Mack Street	Xenia
7	Smith	Jane	123 Main Street	Toledo
8	Doe	John	123 Mack Street	Toledo
9	Jones	John	123 Mean Street	Toledo

FIGURE 15-1

NOTE There is an error bypass method in some of these macros that might be unfamiliar to you. Lesson 20 covers the topic of error handling.

```
Sub DeleteDuplicateRecords()
    'Turn off ScreenUpdating to speed up the macro.
    Application.ScreenUpdating = False

    'Declare a range variable for the helper column being used.
    Dim FilterRange As Range
    'Define the range variable's dynamic range.
    Set FilterRange = Range("E1:E" & Cells(Rows.Count, 1).End(xlUp).Row)

    'For efficiency, open a With structure for the FilterRange variable.
    With FilterRange
```

```

'Enter the formula
'=SUMPRODUCT((A$1:$A1=$A1)*($B$1:$B1=$B1)*($C$1:$C1=$C1)*($D$1:$D1=$D1))>1
'in all cells in column E (the helper column) that returns either TRUE
;if the record is a duplicate of a previous one, or FALSE if the record
'is unique among the records in all previous rows in the list.
.FormulaR1C1 =
"=SUMPRODUCT((R1C1:RC1=RC1)*(R1C2:RC2=RC2)*(R1C3:RC3=RC3)*(R1C4:RC4=RC4))>1"
'Turn the formulas into static values because they will be filtered,
'and maybe deleted if any return TRUE.
.Value = .Value
'AutoFilter the helper column for TRUE.
.AutoFilter Field:=1, Criteria1:="TRUE"
'Error bypass in case no TRUES exist in the helper column.
On Error Resume Next
'This next line resizes the FilterRange variable to exclude the first row.
'Then, it deletes all visible filtered rows.
.Offset(1).Resize(.Rows.Count - 1).SpecialCells(xlCellTypeVisible).EntireRow.Delete
'Clear the Error object in case a run time error would have occurred,
'that is, if no TRUES existed in the helper column to be deleted.
Err.Clear
'Close the With structure for the FilterRange variable object.
End With

'Exit (stop using) AutoFilter.
ActiveSheet.AutoFilterMode = False

'Clear all helper values (there would only be FALSEs at this moment).
'Note that Columns(5) means column E which is the fifth column from the left
'on an Excel spreadsheet.
Columns(5).Clear

'Clear the range object variable to restore system memory.
Set FilterRange = Nothing

'Turn ScreenUpdating back on.
Application.ScreenUpdating = True

End Sub

```

Deleting Some Duplicates and Keeping Others

This section shows a “this way or that way” pair of macros that use an array to hold a set of items to determine which rows you want to keep or delete. In Figure 15-2, an original list has clothing items in column A that are accompanied by various colors of those items in column B.

Both macros hold the same array items of Red, White, and Blue. The macro named `KeepOnlyArrayColors` keeps all rows where Red, White, or Blue are found in column B, while deleting all the other rows. The macro named `DeleteArrayColors` does the opposite: It deletes all rows where Red, White, or Blue are found in column B, but keeps all the other rows.

Original list with colors in column B.			Keep only rows with Red, White, or Blue in column B.			Delete rows having Red, White, or Blue in column B.		
	A	B		A	B		A	B
1	Item	Color	1	Item	Color	1	Item	Color
2	Shirts	White	2	Pants	Red	2	Dresses	Green
3	Pants	Red	3	Shirts	Blue	3	Hats	Black
4	Dresses	Green	4	Pants	White	4	Shoes	Green
5	Hats	Black	5	Hats	Blue	5	Belts	Yellow
6	Shoes	Green	6	Belts	Blue	6	Dresses	Green
7	Belts	Yellow	7	Shirts	Red	7	Shoes	Brown
8	Shirts	Blue	8	Dresses	Blue	8	Pants	Yellow
9	Pants	White	9	Hats	Red	9	Shirts	Brown
10	Dresses	Green	10	Shoes	Blue	10	Pants	Black
11	Hats	Blue	11	Belts	Blue	11	Dresses	Green
12	Shoes	Brown	12	Hats	Blue	12	Shoes	Yellow
13	Belts	Blue	13	Belts	White	13	Pants	Black
14	Shirts	Red	14	Shirts	Red	14		
15	Pants	Yellow	15	Dresses	Blue	15		
16	Dresses	Blue	16	Hats	Blue	16		
17	Hats	Red	17	Shoes	Red	17		
18	Shoes	Blue	18			18		
19	Belts	Blue	19			19		
20	Shirts	Brown	20			20		
21	Pants	Black	21			21		
22	Dresses	Green	22			22		
23	Hats	Blue	23			23		
24	Shoes	Yellow	24			24		
25	Belts	White	25			25		
26	Shirts	Red	26			26		
27	Pants	Black	27			27		
28	Dresses	Blue	28			28		
29	Hats	Blue	29			29		
30	Shoes	Red	30			30		

FIGURE 15-2

```

Sub KeepOnlyArrayColors()
Application.ScreenUpdating = False
Dim LastRow as Long, rng As Range
LastRow = Cells(Rows.Count, 1).End(xlUp).Row
Set rng = Range("B2:B" & LastRow)
Dim ColorList As Variant, ColorItem As Variant
ColorList = Array("Red", "White", "Blue")
For Each ColorItem In ColorList
    rng.Replace What:="ColorItem", Replacement:=ColorItem & "|", LookAt:=xlWhole
    Next ColorItem
    rng.AutoFilter Field:=1, Criteria1:="<>*|"
On Error Resume Next
rng.SpecialCells(xlCellTypeVisible).EntireRow.Delete
Err.Clear
rng.Replace What:="|", Replacement:"", LookAt:=xlPart
Set rng = Nothing
ActiveSheet.AutoFilterMode = False

```

```

Application.ScreenUpdating = True
End Sub

Sub DeleteArrayColors()
Application.ScreenUpdating = False
Dim LastRow as Long, rng As Range
LastRow = Cells(Rows.Count, 1).End(xlUp).Row
Set rng = Range("B2:B" & LastRow)
Dim ColorList As Variant, ColorItem As Variant
ColorList = Array("Red", "White", "Blue")
For Each ColorItem In ColorList
    rng.Replace What:=ColorItem, Replacement:="", LookAt:=xlWhole
Next ColorItem
On Error Resume Next
rng.SpecialCells(xlCellTypeBlanks).EntireRow.Delete
Err.Clear
Set rng = Nothing
Application.ScreenUpdating = True
End Sub

```

WORKING WITH DUPLICATE DATA

As I wrote at the beginning of this lesson, the nature of some projects is to expect duplicated data and to work with it in some way. The following examples show how VBA can make duplicated data work to your advantage.

Compiling a Unique List from Multiple Columns

From a single-column list containing repeated items, you can extract a list of unique items using `AdvancedFilter`. For example, the following line of code copies a unique list of items from column A into column B:

```

Range("A1").CurrentRegion.AdvancedFilter Action:=xlFilterCopy, _
CopyToRange:=Range("B1"), Unique:=True

```

The question becomes, what if you want to extract a unique list from a table that has many columns of repeatedly listed items? In Figure 15-3, a fictional quarterly survey ranks the top-10 vacation destinations. Many of those destinations are repeated among the four quarterly columns. The macro named `UniqueList` lists all unique vacation destinations from the table in column G:

```

Sub UniqueList()
'Turn off ScreenUpdating
Application.ScreenUpdating = False

'Declare and define variables
Dim cell As Range, TableRange As Range
Dim xRow As Long, varCell As Variant
Set TableRange = Range("B4:E13")
xRow = 2

'Clear column G (column #7) where the unique list will go.
Columns(7).Clear

```

```

'Enter the header label in cell G1 and bold cell G1.
With Range("G1")
    .Value = "Unique list:"
    .Font.Bold = True
End With

'Loop through each cell in the table range,
'and add that cell's value to the list if it
'does not exist in the list yet.
For Each cell In TableRange
    varCell = Application.Match(cell.Value, Columns(7), 0)
    If IsError(varCell) Then
        Err.Clear
        Cells(xRow, 7).Value = cell.Value
        xRow = xRow + 1
    End If
    Next cell

'Clear the TableRange object variable from system memory.
Set TableRange = Nothing

'Optional, sort the list in alphabetical order.
Range("G1").CurrentRegion.Sort Key1:=Range("G2"), _
    Order1:=xlAscending, Header:=xlYes
'Autofit column G.
Columns(7).AutoFit

'Turn ScreenUpdating back on.
Application.ScreenUpdating = True
End Sub

```

	A	B	C	D	E	F	G
1	Survey of Favorite Vacation Destinations					Unique list:	
2						Africa	
3	Rank	Quarter 1	Quarter 2	Quarter 3	Quarter 4	Australia	
4	1	New York	San Francisco	San Francisco	Disney World	Bahamas	
5	2	San Francisco	New York	Paris	San Francisco	Brazil	
6	3	Paris	Paris	London	New York	Cancun	
7	4	Disney World	Italy	Cancun	Bahamas	Caribbean	
8	5	London	Disney World	Africa	Paris	Disney World	
9	6	Cancun	Las Vegas	New York	Brazil	Hawaii	
10	7	Tokyo	Hawaii	Quebec	London	Italy	
11	8	Bahamas	Australia	Bahamas	Hawaii	Las Vegas	
12	9	Africa	Caribbean	Tokyo	Australia	London	
13	10	Las Vegas	Quebec	Australia	Caribbean	New York	
14					Paris		
15					Quebec		
16					San Francisco		
17					Tokyo		
18							

FIGURE 15-3

Updating a Comment to List Unique Items

This section shows how you can automatically update a comment to show unique items in sorted order from a list containing repeated items. When a new unique item is added to the list, the comment is immediately updated in real time.

In Figure 15-4, a company keeps an ongoing list of its clients and dates of transactions. When a new client is added to the list, such as what is happening in cell A20, the comment in cell A1 is updated to show that new client name in a sorted list.

Before change		During change		After change	
A	B	A	B	A	B
1 Client	Date	1 Get Plastered	Aug 1, 2015	2 Get Plastered	Aug 1, 2015
2 Get Plastered	Aug 1, 2015	2 Wok and Roll	Aug 1, 2015	3 Wok	Unique client names: , 2015
3 Wok	Unique client names: , 2015	3 Sew What	Aug 1, 2015	4 Sew	, 2015
4 Sew	, 2015	4 Buy the Book	Aug 1, 2015	5 Buy t	Buy the Book , 2015
5 Buy t	Buy the Book , 2015	5 Get Plastered	Aug 2, 2015	6 Buy t	Drain Surgeons , 2015
6 Wok	Sew What , 2015	6 Wok and Roll	Aug 2, 2015	6 Wok	Get Plastered , 2015
7 Get F	Wok and Roll , 2015	7 Get Plastered	Aug 2, 2015	7 Get F	Sew What , 2015
8 Buy the Book	Aug 2, 2015	8 Buy the Book	Aug 2, 2015	8 Buy F	Wok and Roll , 2015
9 Buy the Book	Aug 2, 2015	9 Buy the Book	Aug 2, 2015	9 Buy the Book	Aug 2, 2015
10 Sew What	Aug 3, 2015	10 Sew What	Aug 3, 2015	10 Sew What	Aug 3, 2015
11 Wok and Roll	Aug 3, 2015	11 Wok and Roll	Aug 3, 2015	11 Wok and Roll	Aug 3, 2015
12 Buy the Book	Aug 3, 2015	12 Buy the Book	Aug 3, 2015	12 Buy the Book	Aug 3, 2015
13 Sew What	Aug 3, 2015	13 Sew What	Aug 3, 2015	13 Sew What	Aug 3, 2015
14 Sew What	Aug 3, 2015	14 Sew What	Aug 3, 2015	14 Sew What	Aug 3, 2015
15 Get Plastered	Aug 3, 2015	15 Get Plastered	Aug 3, 2015	15 Get Plastered	Aug 3, 2015
16 Wok and Roll	Aug 3, 2015	16 Wok and Roll	Aug 3, 2015	16 Wok and Roll	Aug 3, 2015
17 Buy the Book	Aug 4, 2015	17 Buy the Book	Aug 4, 2015	17 Buy the Book	Aug 4, 2015
18 Get Plastered	Aug 4, 2015	18 Get Plastered	Aug 4, 2015	18 Get Plastered	Aug 4, 2015
19 Get Plastered	Aug 4, 2015	19 Get Plastered	Aug 4, 2015	19 Get Plastered	Aug 4, 2015
20		20 Drain Surg		20 Drain Surgeons	Aug 7, 2015
21		21		21	

FIGURE 15-4

NOTE This example uses a Worksheet_Change event procedure. The code goes into the module of your worksheet. Lesson 13 covers event coding, including how and where to place this code.

```
Private Sub Worksheet_Change(ByVal Target As Range)
    'Limit the event to monitor only changes in column A.
    If Target.Column <> 1 Then Exit Sub

    'Prepare Excel's application settings.
    With Application
        .ScreenUpdating = False
        .DisplayAlerts = False
    End With
End Sub
```

```
.EnableEvents = False

'Declare variables.
Dim HelperColumn As Long, cell As Range, strCommentText As String

'Define the helper column which is the last used column + 2,
'to use for listing the unique client names and sorting them.
HelperColumn = _
Cells.Find(What:="*", After:=Range("A1"), _
SearchOrder:=xlByColumns, _
SearchDirection:=xlPrevious).Column + 2

'List the unique client names in the helper column.
Range("A1:A" & Cells(Rows.Count, 1).End(xlUp).Row).AdvancedFilter _
Action:=xlFilterCopy, CopyToRange:=Cells(1, HelperColumn), Unique:=True

'Sort the unique client list in ascending order.
Cells(1, HelperColumn).Sort _
Key1:=Cells(2, HelperColumn), _
Order1:=xlAscending, _
Header:=xlYes

'Build the comment's text string, comprised by each unique client name
'in a vertical list. To do that, separate each name with the ascii 10
'carriage return character.
strCommentText = ""
For Each cell In Cells(1, HelperColumn).CurrentRegion
'Bypass the header cell in row 1.
If cell.Row <> 1 Then _
strCommentText = strCommentText & Chr(10) & cell.Value
Next cell
strCommentText = "Unique client names:" & Chr(10) & strCommentText

'You are maintaining your comment in cell A1 that lists the unique
'client names whenever a new one is added to column A in the table.
With Range("A1")
If Not .Comment Is Nothing Then .Comment.Delete
.AddComment
With .Comment
.Visible = False
.Text Text:=strCommentText
.Shape.TextFrame.AutoSize = True
End With
End With

'Clear the helper column's unique list which now is represented
'in the comment.
Columns(HelperColumn).Clear

'Reset Excel's application settings.
.EnableEvents = True
.DisplayAlerts = True
.ScreenUpdating = True
End With

End Sub
```

Selecting a Range of Duplicate Items

This section shows a convenient way to select a range of cells with duplicate items in a column. In this example, Figure 15-5 shows a list that is sorted by column A. When you double-click any cell in the table, rows are selected that have the same item in column A as the cell in column A of the row you double-clicked.

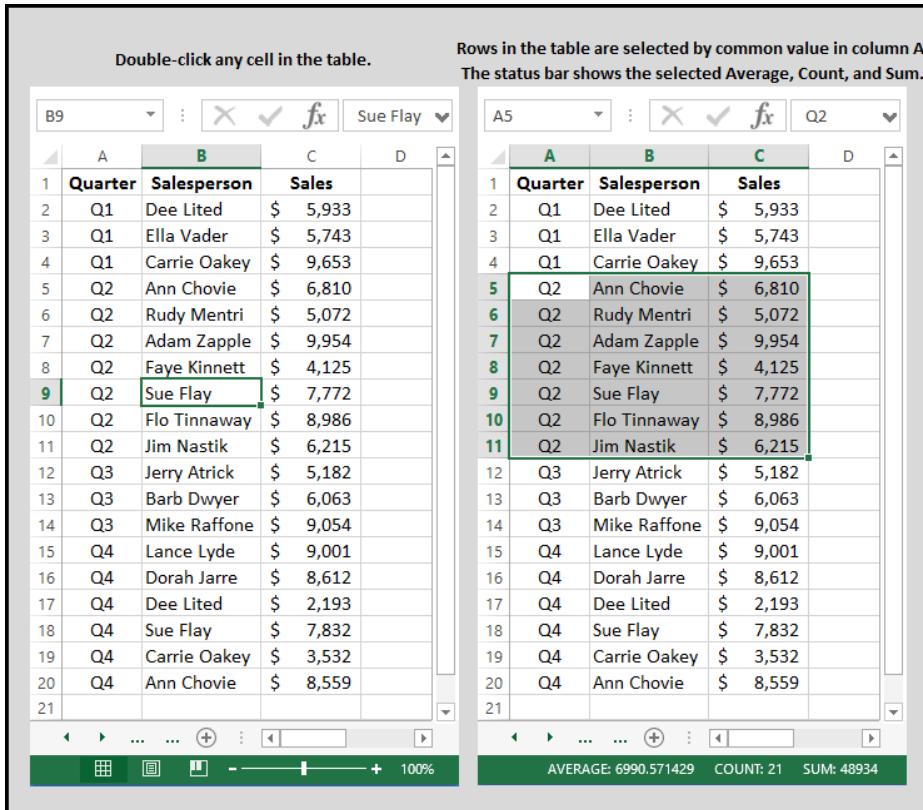


FIGURE 15-5

NOTE Although this example shows the Select method, you can change the code to a different method, such as to copy or format the range.

One of the conveniences of selecting the relevant range, as shown in Figure 15-5, is to quickly view the selection's calculated information on the status bar. This is a worksheet-level event procedure, so the following code goes into the module of your worksheet:

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

```
'Program only for rows in the list, excluding row 1.  
If Target.Row = 1 Then Exit Sub  
If Intersect(Target, Range("A1").CurrentRegion) Is Nothing Then Exit Sub  
Cancel = True  
  
'Declare variables  
Dim myVal As String, LastColumn As Long  
Dim Add1 As Long, Add2 As Long  
Dim xRow As Long, LastRow As Long  
  
'Define variables  
myVal = Cells(Target.Row, 1).Value  
LastRow = Cells(Rows.Count, 1).End(xlUp).Row  
LastColumn =  
Cells.Find(What:="*", After:=Range("A1"), SearchOrder:=xlByColumns, _  
SearchDirection:=xlPrevious).Column  
Add1 = Columns(1).Find(What:=myVal, LookIn:=xlValues, LookAt:=xlWhole).Row  
xRow = Add1  
  
'Identify the range of rows having the same values in column A.  
Do  
  
If Cells(xRow + 1, 1).Value <> myVal Then  
Add2 = xRow + 1  
Exit Do  
Else  
xRow = xRow + 1  
End If  
Loop Until xRow = LastRow  
Add2 = xRow  
  
'Select (or copy or format) records having the same values in column A.  
Range(Cells(Add1, 1), Cells(Add2, LastColumn)).Select  
End Sub
```

Inserting an Empty Row at Each Change in Items

A common request is how to insert an empty row at each change of data in a column. In Figure 15-6, a table is preferred to be sorted by the Client Name column with an empty row at each change in Client Name. The macro named `Sort_Separate_ClientName` does that, with comments along the way to explain the process:

```
Sub Sort_Separate_ClientName()  
'Turn off ScreenUpdating.  
Application.ScreenUpdating = False  
'Sort the table by ClientName in ascending order.  
Range("A3").CurrentRegion.Sort  
Key1:=Range("A4"), Order1:=xlAscending, Header:=xlYes  
'Declare a Long type variable for the last row in column A.  
Dim LastRow As Long  
'Determine the last row of data.  
LastRow = Cells(Rows.Count, 1).End(xlUp).Row  
'Declare a Long type variable for evaluating each row.  
Dim xRow As Long
```

```
'Loop through each ClientName item in column A of the table.
'When the item being evaluated is not the same as the item
'in the row above it, that means the client name is different.
'Insert an empty row at that change.
'Notice, work from the bottom row upwards because you are
'inserting rows.
For xRow = LastRow To 5 Step -1
If Cells(xRow, 1).Value <> Cells(xRow - 1, 1).Value Then _
Rows(xRow).Resize(1).Insert
Next xRow
'Turn ScreenUpdating on again.
Application.ScreenUpdating = True
End Sub
```

Original list, unsorted and contiguous.

	A	B	C
1	Client Transactions		
2			
3	Client Name	Sales Date	Sales
4	Media Moguls	Jun 3, 2015	\$9,795
5	Jack's Lumber	Sep 6, 2015	\$8,485
6	Media Moguls	May 13, 2015	\$9,102
7	ACME Corp	Feb 4, 2015	\$5,632
8	CyberBots, Inc	Jul 11, 2015	\$8,539
9	Media Moguls	Aug 26, 2015	\$5,484
10	CyberBots, Inc	Apr 25, 2015	\$4,705
11	Jack's Lumber	Oct 1, 2015	\$5,156
12	ACME Corp	Jun 7, 2015	\$7,265
13	Buy The Book	Nov 30, 2015	\$5,434
14	Jack's Lumber	Oct 9, 2015	\$4,135
15	Gadgets-R-Us	Sep 17, 2015	\$6,287
16	Buy The Book	Aug 23, 2015	\$9,423
17	Gadgets-R-Us	Jan 25, 2015	\$6,584
18	CyberBots, Inc	Mar 16, 2015	\$1,841
19	Sunny Daze Co	Jan 8, 2015	\$2,062
20	Sunny Daze Co	Mar 30, 2015	\$2,910
21	Media Moguls	Feb 20, 2015	\$ 952
22	Sunny Daze Co	Apr 18, 2015	\$8,822
23			
24			
25			
26			
27			
28			

Sorted list, with empty row at each change in Client.

	A	B	C
1	Client Transactions		
2			
3	Client Name	Sales Date	Sales
4	ACME Corp	Feb 4, 2015	\$5,632
5	ACME Corp	Jun 7, 2015	\$7,265
6			
7	Buy The Book	Nov 30, 2015	\$5,434
8	Buy The Book	Aug 23, 2015	\$9,423
9			
10	CyberBots, Inc	Jul 11, 2015	\$8,539
11	CyberBots, Inc	Apr 25, 2015	\$4,705
12	CyberBots, Inc	Mar 16, 2015	\$1,841
13			
14	Gadgets-R-Us	Sep 17, 2015	\$6,287
15	Gadgets-R-Us	Jan 25, 2015	\$6,584
16			
17	Jack's Lumber	Sep 6, 2015	\$8,485
18	Jack's Lumber	Oct 1, 2015	\$5,156
19	Jack's Lumber	Oct 9, 2015	\$4,135
20			
21	Media Moguls	Jun 3, 2015	\$9,795
22	Media Moguls	May 13, 2015	\$9,102
23	Media Moguls	Aug 26, 2015	\$5,484
24	Media Moguls	Feb 20, 2015	\$ 952
25			
26	Sunny Daze Co	Jan 8, 2015	\$2,062
27	Sunny Daze Co	Mar 30, 2015	\$2,910
28	Sunny Daze Co	Apr 18, 2015	\$8,822

FIGURE 15-6

TRY IT

For this lesson, a table of data includes names of stores in column A that are repeated elsewhere in the column. A macro is requested to copy the individual rows of data for each unique store name, and paste those rows into their own workbook.

The workbooks are named by the name of the store, appended with the date and time the macro was run. The workbooks are saved in the same folder path as the workbook holding the original data.

Lesson Requirements

To get the sample workbook, you can download Lesson 15 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

The following hints might help you as you complete this Try It:

- Your list of data need not be too lengthy; a couple dozen rows of data would suffice.
- In the downloadable workbook for this lesson, column A contains a list of store names, which is why you see the references to "Store" in the Step-by-Step code.
- Repeat each item at least once (as mentioned in Step 1), but feel free to repeat each item as many times in column A as you want.
- For convenience, the destination path where the new workbooks will be saved is the same as the path of the workbook holding the original data.
- When you use a helper column or row, be sure to leave at least one empty column or row between it and the data table you are working with. Without the empty column or row, VBA might assume your helper data is a part of the original table.
- When your macros involve creating or working in other workbooks while you refer to a worksheet in your workbook holding the macro, be sure to qualify your worksheet's parent name with the `ThisWorkbook` object.
- When your macros create potentially dozens or hundreds of new workbooks, close the workbooks after you name them as shown in Step 24. It's rare for a user to want that many workbooks open at the same time after the macro has completed.

Step-by-Step

1. Start by opening a new workbook and copy or enter a table of data that includes a few columns. Put column labels in row 1, and repeat each of the entries in column A at least once.
2. Save your workbook as a macro-enabled type with the extension `.xlsm`.
3. Press Alt+F11 to go to the Visual Basic Editor.
4. From the VBE menu bar, click Insert \Rightarrow Module.
5. In the module you just created, type `Sub UniqueStoresToWorkbooks` and press Enter. VBA automatically places a pair of empty parentheses at the end of the `Sub` line, followed by an empty line, and the `End Sub` line below that. Your macro should look like this so far:

```
Sub UniqueStoresToWorkbooks()
```

```
End Sub
```

6. Turn off ScreenUpdating to speed up the macro when you run it, and to keep your screen from flickering, which happens during macros that manipulate row, column, and workbook objects as this macro does:

```
Application.ScreenUpdating = False
```

7. Declare variables:

```
'Identify and count each row of a unique list of items in column A.  
Dim UniqueRow As Long, lngUniqueCount As Long  
'String variables for each unique item name and its workbook name.  
Dim strUniqueStore As String, strUniqueStoreWbname As String  
'Number of the data table's last row; next available column one column removed  
'from the rightmost column of the data table; range occupied by the data table.  
Dim LastRow As Long, NextColumn As Long, FilterRange As Range  
'Path to receive the new workbooks; name of sheet where the data table resides.  
Dim strDestinationFolderPath As String, asn As String
```

8. Define the destination path that will receive the new workbooks, which is the same path of the active workbook holding this macro:

```
strDestinationFolderPath = ThisWorkbook.Path & "\\"
```

9. Define the sheet name holding the original list:

```
asn = ActiveSheet.Name
```

10. Identify the last row in the list, using column A:

```
LastRow = Cells(Rows.Count, 1).End(xlUp).Row
```

11. Identify the column that is two columns removed from the right-most column in the list. This column will hold the unique store names, with one empty column separating it from the list:

```
NextColumn = Cells.Find(What:="*", After:=Range("A1"), _  
SearchOrder:=xlByColumns, SearchDirection:=xlPrevious).Column + 2
```

12. Define the range (which is column A of the list) that will be filtered for each unique store name:

```
Set FilterRange = _  
ThisWorkbook.Worksheets(asn).Range("A1:A" & LastRow)
```

13. List all unique store names using AdvancedFilter:

```
FilterRange.AdvancedFilter _  
Action:=xlFilterCopy, CopyToRange:=Cells(1, NextColumn), Unique:=True
```

14. Count the unique store names, not including the header cell. This is a service to the users to let them know in a message box at the end of the macro how many unique items were found, hence how many new workbooks were created:

```
lngUniqueCount = WorksheetFunction.CountA(Columns(NextColumn)) - 1
```

15. Open a For...Next loop to loop through all unique store names to be filtered for exposing their respective data:

```
For UniqueRow = 2 To Cells(Rows.Count, NextColumn).End(xlUp).Row
```

16. Create the workbook to hold the next unique store name. The 1 in this syntax refers to a standard Excel worksheet:

```
Workbooks.Add 1
```

17. Assign the name of the next unique store to the strUniqueStore variable. Turn off AutoFilter first to expose all rows on the worksheet:

```
With ThisWorkbook.Worksheets(asn)
    .AutoFilterMode = False
    strUniqueStore = .Cells(UniqueRow, NextColumn).Value
End With
```

18. Define the full workbook name of the next unique store name, including the extension. The workbook name's date and time suffix helps to reference the creation date at a glance when the workbooks are viewed in Windows File Explorer, and to avoid overriding existing workbook names:

```
strUniqueStoreWBname = strUniqueStore & "_" &
Format(VBA.Now, "YYYYMMDD_HHMMSS") & ".xlsx"
```

19. AutoFilter the list for the next unique store name:

```
FilterRange.AutoFilter Field:=1, Criteria1:=strUniqueStore
```

20. Copy the visible (filtered) rows for this unique store name, and paste them to the workbook you created for it in Step 16:

```
FilterRange.SpecialCells(xlCellTypeVisible).EntireRow.Copy Range("A1")
```

21. Keep in mind that the active workbook at this moment is the new workbook you created for it. The unique list of store names is still visible and not wanted, so clear that column:

```
Columns(NextColumn).Clear
```

22. Autofit the columns in this new workbook for readability as a service to the user:

```
Cells.Columns.AutoFit
```

23. Save the new workbook:

```
ActiveWorkbook.SaveAs _
Filename:=strDestinationFolderPath & _
strUniqueStoreWBname, FileFormat:=51
```

NOTE In Step 18, the workbooks are saved with the .xlsx extension, which is why the statement FileFormat:=51 is required when naming the files. If you save a workbook with the .xlsm extension, the statement FileFormat:=52 would be required.

24. Close the new workbook:

```
ActiveWorkbook.Close
```

- 25.** Continue the loop for all the unique store names:

```
Next UniqueRow
```

- 26.** Reactivate this workbook and the worksheet holding the original data table:

```
ThisWorkbookActivate  
Worksheets(asn).Activate
```

- 27.** Turn off AutoFilter:

```
ActiveSheet.AutoFilterMode = False
```

- 28.** Clear the unique list that you created in Step 13:

```
Columns(NextColumn).Clear
```

- 29.** Release the FilterRange object variable from system memory:

```
Set FilterRange = Nothing
```

- 30.** Turn ScreenUpdating back on:

```
Application.ScreenUpdating = True
```

- 31.** With a message box, confirm for the user that the task is completed:

```
MsgBox _  
"There were " & lngUniqueCount & " different Stores." & vbCrLf & _  
"Their respective data has been consolidated into" & vbCrLf & _  
"individual workbooks, all saved in the path" & vbCrLf & _  
strDestinationFolderPath & ".", vbInformation, "Done!"  
End Sub
```

- 32.** With your macro completed, press Alt+Q to return to the worksheet. To test the macro, press Alt+F8 to show the Macro dialog box. Select the macro named UniqueStoresToWorkbooks and click Run. Here is what the macro looks like in its entirety:

```
Sub UniqueStoresToWorkbooks()  
  
'Turn off screen updating.  
Application.ScreenUpdating = False  
  
'Declare and define variables.  
'Identify and count each row of a unique list of items in column A.  
Dim UniqueRow As Long, lngUniqueCount As Long  
'String variables for each unique item name and its workbook name.  
Dim strUniqueStore As String, strUniqueStoreWBname As String  
'Number of the data table's last row; next available column one column removed  
'from the rightmost column of the data table; range occupied by the data table.  
Dim LastRow As Long, NextColumn As Long, FilterRange As Range  
'Path to receive the new workbooks; name of sheet where the data table resides.  
Dim strDestinationFolderPath As String, asn As String  
  
'Define variables.  
'The destination path that will receive these new workbooks  
'is the same path as the active workbook.  
strDestinationFolderPath = ThisWorkbook.Path & "\\"  
'Start from the sheet name holding the original list.
```

```
asn = ActiveSheet.Name
'Identify the last cell row of the data table.
LastRow = Cells(Rows.Count, 1).End(xlUp).Row
'Identify the column that is 2 columns removed from
'the right-most column in the list.
NextColumn = Cells.Find(What:="*", After:=Range("A1"), _
SearchOrder:=xlByColumns, _
SearchDirection:=xlPrevious).Column + 2
'The range (which is column A of the list) that will be
'filtered for each unique store name.
Set FilterRange =
ThisWorkbook.Worksheets(asn).Range("A1:A" & LastRow)

>List all unique Store Names using AdvancedFilter.
FilterRange.AdvancedFilter Action:=xlFilterCopy, _
CopyToRange:=Cells(1, NextColumn), Unique:=True

'Count the unique Stores, not including the header cell.
'This is a service to the user to let them know in a message box
'at the end of the macro how many unique items were found,
'meaning how many new workbooks were created.
lngUniqueCount = WorksheetFunction.CountA(Columns(NextColumn)) - 1

'Open a For...Next loop, to loop through all
'unique store names, filter for them, and paste their data to a
'new workbook, saved with creation date and time.
For UniqueRow = 2 To Cells(Rows.Count, NextColumn).End(xlUp).Row

>Create the workbook to hold the next unique store name.
Workbooks.Add 1

'Assign the name of the next unique store to
'the strUniqueStore variable.
'AutoFilter is turned off first to expose all rows on the sheet.
With ThisWorkbook.Worksheets(asn)
.AutoFilterMode = False
strUniqueStore = .Cells(UniqueRow, NextColumn).Value
End With

'Define the full workbook name of the next
'unique store name, including extension.
'The workbook name's date and time suffix helps to
'reference the creation date at a glance when the
'workbooks are viewed in Windows File Explorer,
'and to avoid overriding existing workbook names.
strUniqueStoreWBname = strUniqueStore & "_" & _
Format(VBA.Now, "YYYYMMDD_HHMMSS") & ".xlsx"

'Filter the list for that next unique store name.
FilterRange.AutoFilter Field:=1, Criteria1:=strUniqueStore

'Copy the visible (filtered) rows for this unique
'store name, and paste them to their new workbook.
FilterRange.SpecialCells(xlCellTypeVisible).EntireRow.Copy Range("A1")
```

```
'Keep in mind that the active workbook at this moment
'is the new workbook you created for it. The unique list of
'store names is still visible and not needed, so clear that column.
Columns(NextColumn).Clear
'Autofit the columns in this new workbook for readability.
Cells.Columns.AutoFit

'Save and close the new workbook.
ActiveWorkbook.SaveAs _
Filename:=strDestinationFolderPath & _
strUniqueStoreWbname, FileFormat:=51
'Close the new workbook.
ActiveWorkbook.Close

'Continue the loop through all the unique store names.
Next UniqueRow

'Re-activate this workbook and the source worksheet.
ThisWorkbook.Activate
Worksheets(asn).Activate
'Turn off autofilter.
ActiveSheet.AutoFilterMode = False
'Clear the unique list.
Columns(NextColumn).Clear

'Release the object variable from system memory.
Set FilterRange = Nothing
'Turn screen updating back on.
Application.ScreenUpdating = True

'Confirm for the user that the parsing is completed.
MsgBox _
"There were " & lngUniqueCount & " different Stores." _
& vbCrLf & _
"Their respective data has been consolidated into" & _
vbCrLf & _
"individual workbooks, all saved in the path" & vbCrLf & _
strDestinationFolderPath & ".", vbInformation, "Done!"

End Sub
```

REFERENCE Please select the video for Lesson 15 at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

16

Using Embedded Controls

You've seen many ways to run macros, including using keyboard shortcuts, the Macro dialog box, and the Visual Basic Editor. This lesson shows you how to execute VBA code by clicking a button or other object that you can place onto your worksheet to make your macros easier to run.

WORKING WITH FORM CONTROLS AND ACTIVEX CONTROLS

A *control* is an object such as a Button, Label, TextBox, OptionButton, or CheckBox that you can place onto a UserForm (covered in Lessons 21, 22, and 23) or embed onto a worksheet. VBA supports these and more controls, which provide an intuitive way for you to run your macros quickly and with minimal effort.

Excel supports two generations of controls. *Form* controls are the original controls that came with Excel, starting with version 5. Form controls are still fully supported in all later versions of Excel, including Excel 2013. Form controls are more stable, simpler to use, and more integrated with Excel. For example, you can place a Form control onto a chart sheet, but you cannot do that with an ActiveX control.

Generally, ActiveX controls from the Control Toolbox are more flexible with their extensive properties and events. You can customize their appearance, behavior, fonts, and other characteristics. You can also control how different events are responded to when an ActiveX control is associated with those events.

Form controls have macros that are assigned to them. ActiveX controls run procedures that are based on whatever event(s) they have been programmed to monitor. ActiveX controls don't look all that scintillating, but Form controls have an even more elementary appearance that will never win them first prize in a beauty contest. However, both kinds of controls serve their purposes well as Microsoft intended, and they are here to stay with Excel for the foreseeable future.

CHOOSING BETWEEN FORM CONTROLS AND ACTIVEX CONTROLS

The primary differences between the two kinds of controls are in formatting and events. You use Form controls when you need simple interaction with VBA, such as running a macro by clicking a button. They are also a good choice when you don't need VBA at all, but you want an option button or check box on your sheet that will be linked to a cell. If you need to color your control, or format its font type, or trigger a procedure based on mouse movement or keyboard activity, ActiveX controls are the better choice.

Be aware that ActiveX controls have a well-deserved reputation for being buggy and not behaving as reliably as do Form controls. Form controls will give you minimal problems, if any, but they are limited in what they can do. As you experiment and work with each type, you'll decide which kind of control works best for your purposes.

The Forms Toolbar

The easiest way to access Form controls is through the Forms toolbar. How you get to the Forms toolbar depends on your version of Excel. For versions prior to Excel 2007, from the worksheet menu, click View \Rightarrow Toolbars \Rightarrow Forms, as shown in Figure 16-1.

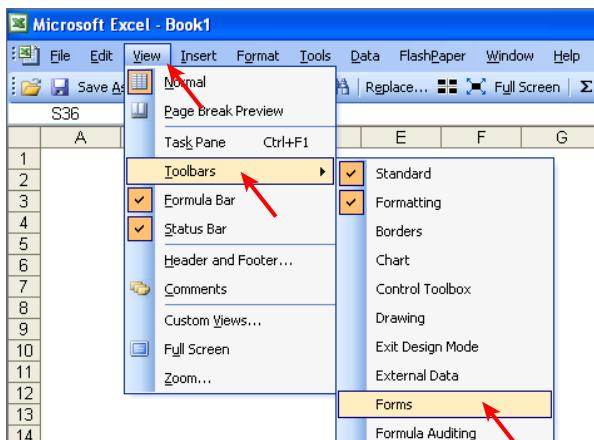


FIGURE 16-1

The Forms toolbar is like any other toolbar that you can dock at the top or sides of the window, or have floating on the window above the worksheet. Figure 16-2 shows the Forms toolbar and its control icons.

If you are using Excel version 2007, 2010, or 2013, you get to the Forms and ActiveX controls by clicking the Insert icon on the Developer tab of the Ribbon, as shown in Figure 16-3.

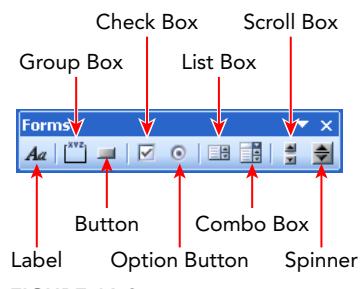
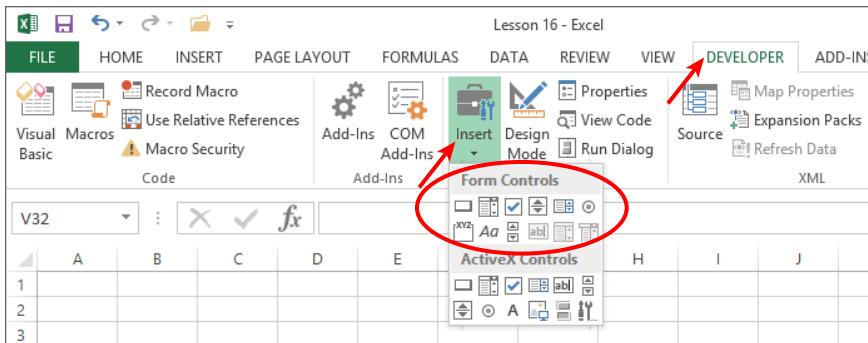


FIGURE 16-2



Form controls in the circled area, first row, left to right:

Button, Combo Box, Check Box, Spin Button, List Box, Option Button

Second row, left to right:

Group Box, Label, Scroll Bar, Text Field, Combo List - Edit, Combo Drop-Down - Edit

FIGURE 16-3

NOTE The Developer tab is a very useful item to place on your Ribbon. See the “Accessing the VBA Environment” section in Lesson 2 for the steps to display the Developer tab.

Buttons

The most commonly used Form control is the Button. When you use a Button, you have a macro in mind that you have either already written or will write, which will be attached to the Button. The following steps are a common sequence of actions when using a Form Button:

1. Create the macro that will be attached to the Button. Suppose you are negotiating rents, and you need to frequently clear the range C4:F4 on a company budget sheet. The macro you'd write is


```
Sub ClearData()
    Range("C4:F4").Clear
End Sub
```
2. To make it easy to run that macro, you can assign it to a Form Button. On the Forms toolbar, click the Button icon. Press down your mouse's left button, then draw the Button into cell B4. As soon as you do, the Assign Macro dialog box appears, as shown in Figure 16-4. Select the macro to be assigned to the Button, and click OK.
3. With your new Button selected, click it and delete the entire default caption. Type the caption **Clear Cells**, as shown in Figure 16-5.
4. Select any worksheet cell to deselect the Button. Go ahead and click the Button to verify that it clears the cells in range C4:F4 as expected.

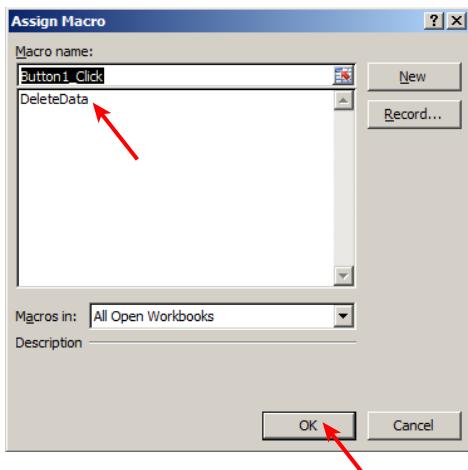


FIGURE 16-4

	A	B	C	D	E	F	G	H
1	Widgets, Inc. Expense Log							
2								
3			Quarter 1	Quarter 2	Quarter 3	Quarter 4	Total	
4	Rent	Clear Cells	\$23,362	\$68,531	\$66,276	\$78,809	\$236,978	
5	Utilities		\$28,166	\$64,728	\$99,216	\$4,160	\$196,270	
6	Payroll		\$55,193	\$97,457	\$24,372	\$85,839	\$262,861	
7	Office Supplies		\$66,540	\$78,889	\$22,349	\$13,606	\$181,384	
8	Maintenance		\$35,135	\$64,505	\$36,173	\$2,033	\$137,846	
9	Landscaping		\$14,088	\$15,934	\$80,263	\$27,142	\$137,427	
10	Total		\$222,484	\$390,044	\$328,649	\$211,589	\$1,152,766	
11								

FIGURE 16-5

Using Application.Caller with Form Controls

One of the cool things about Form controls is that you can apply a single macro to all of them and gain information about which control was clicked. When you know which Button was clicked, you can take a specific action relating to that Button.

Expanding on the previous example, suppose you want to place a Button on each row of data, so that when you click a Button, the cells are cleared in columns C:F of the row where the Button resides.

It's obvious that the original macro applies only to the first Button in the Rent row, so here are the steps to have one macro serve many controls:

1. Modify the `ClearData` macro as follows. For the Button that was clicked, the cell holding that Button's top-left corner is identified. The macro can now be a customization tool for each individual Button to which it is attached:

```
Sub ClearData()
Dim myRow As Long
myRow =
    ActiveSheet.Buttons(Application.Caller).TopLeftCell.Row
Range(Cells(myRow, 3), Cells(myRow, 6)).Clear
End Sub
```

2. Recall that the original macro name is still attached to that Button. Return to your worksheet and right-click the Button. Select **Copy** because you are copying the Button and the macro to which it is attached.
3. Select cell B5 and press **Ctrl+V**. Repeat that step for cells B6, B7, B8, and B9. Your worksheet will resemble Figure 16-6.

	A	B	C	D	E	F	G	H
1	Widgets, Inc. Expense Log							
2								
3			Quarter 1	Quarter 2	Quarter 3	Quarter 4	Total	
4	Rent	Clear Cells	\$23,362	\$68,531	\$66,276	\$78,809	\$236,978	
5	Utilities	Clear Cells	\$28,166	\$64,728	\$99,216	\$4,160	\$196,270	
6	Payroll	Clear Cells	\$55,193	\$97,457	\$24,372	\$85,839	\$262,861	
7	Office Supplies	Clear Cells	\$66,540	\$78,889	\$22,349	\$13,606	\$181,384	
8	Maintenance	Clear Cells	\$35,135	\$64,505	\$36,173	\$2,033	\$137,846	
9	Landscaping	Clear Cells	\$14,088	\$15,934	\$80,263	\$27,142	\$137,427	
10	Total	Clear Cells	\$222,484	\$390,044	\$328,649	\$211,589	\$1,152,766	
11								

FIGURE 16-6

4. Test the macro by clicking the Button on the Office Supplies row. When you click that Button, the macro clears the cells in row 7, columns C:F, as shown in Figure 16-7.

NOTE Attaching a macro to an embedded object is not limited to Form controls. You can attach a macro to pretty much any Drawing shape or picture that you want to embed onto your worksheet.

	A	B	C	D	E	F	G	H
1	Widgets, Inc. Expense Log							
2								
3			Quarter 1	Quarter 2	Quarter 3	Quarter 4	Total	
4	Rent	Clear Cells	\$23,362	\$68,531	\$66,276	\$78,809	\$236,978	
5	Utilities	Clear Cells	\$28,166	\$64,728	\$99,216	\$4,160	\$196,270	
6	Payroll	Clear Cells	\$55,193	\$97,457	\$24,372	\$85,839	\$262,861	
7	Office Supplies	Clear Cells					\$0	
8	Maintenance	Clear Cells	\$35,135	\$64,505	\$36,173	\$2,033	\$137,846	
9	Landscaping	Clear Cells	\$14,088	\$15,934	\$80,263	\$27,142	\$137,427	
10	Total	Clear Cells	\$155,944	\$311,155	\$306,300	\$197,983	\$971,382	
11								

FIGURE 16-7

The Control Toolbox

Similar to the Forms toolbar, the Control Toolbox can be accessed in versions prior to Excel 2007 from the worksheet menu bar. Click View \Rightarrow Toolbars \Rightarrow Control Toolbox, as shown in Figure 16-8.

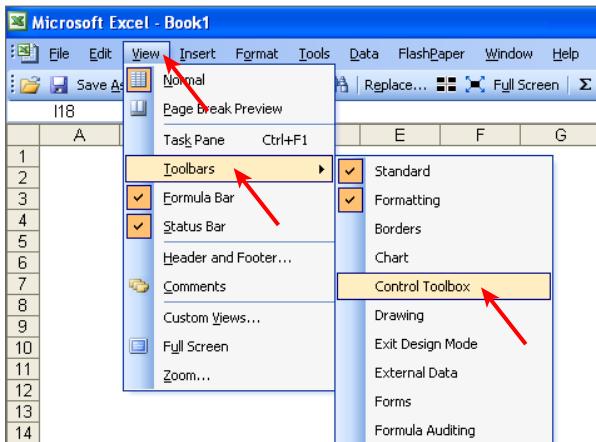


FIGURE 16-8

The Control Toolbox itself is shown in Figure 16-9. If you are using version 2007, 2010, or 2013, you can find the Forms and ActiveX controls by clicking the Insert icon on the Developer tab of the Ribbon, shown in Figure 16-9.

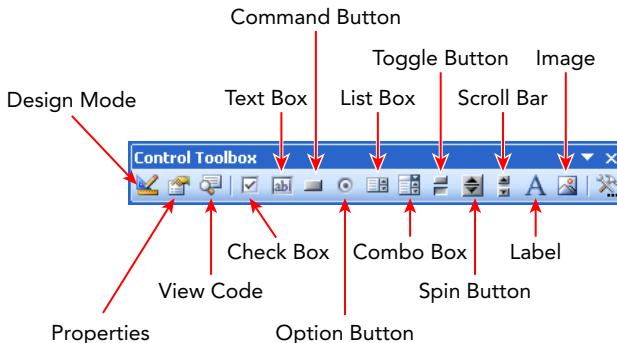
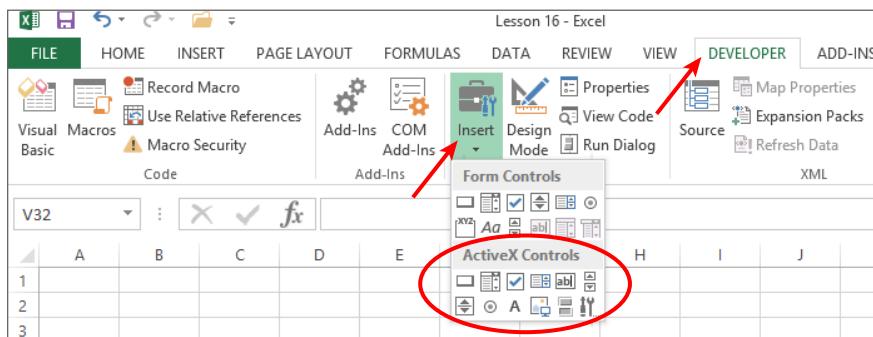


FIGURE 16-9

More than 100 additional ActiveX controls beyond what you see on the Control Toolbox are available. You might notice an icon named More Controls at the far right of the Control Toolbox toolbar, and in the lower-right corner of the Insert icon's drop-down display in Excel 2007, 2010, and 2013. When expanded, that icon (see Figure 16-10), reveals the additional ActiveX controls available for you to embed, as indicated in Figure 16-11.

NOTE *The odds are you'll never need most of those controls, but it gives you a sense of the expansive functionality that is available to you with ActiveX objects.*



ActiveX controls in the circled area, first row, left to right:
 Command Button, Combo Box, Check Box, List Box, Text Box, Scroll Bar
 Second row, left to right:
 Spin Button, Option Button, Label, Image, Toggle Button, More Controls

FIGURE 16-10

CommandButtons

The ActiveX CommandButton is the counterpart to the Form control button. As with virtually every ActiveX object, the CommandButton has numerous properties through which you can customize its appearance. Unlike Form controls, an ActiveX object such as a CommandButton responds to event code. There is no such thing as a macro being attached to a CommandButton.

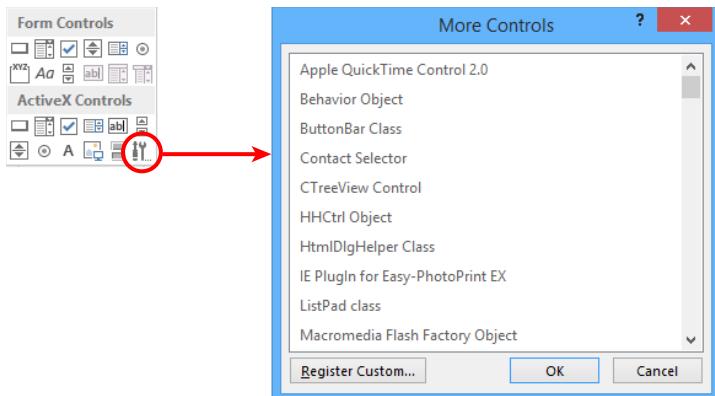


FIGURE 16-11

From the Control Toolbox, draw a CommandButton onto your worksheet. Excel defaults to Design Mode, allowing you to work with the ActiveX object you just created. Right-click the CommandButton and select Properties, as shown in Figure 16-12. You can see the Design Mode icon is active.

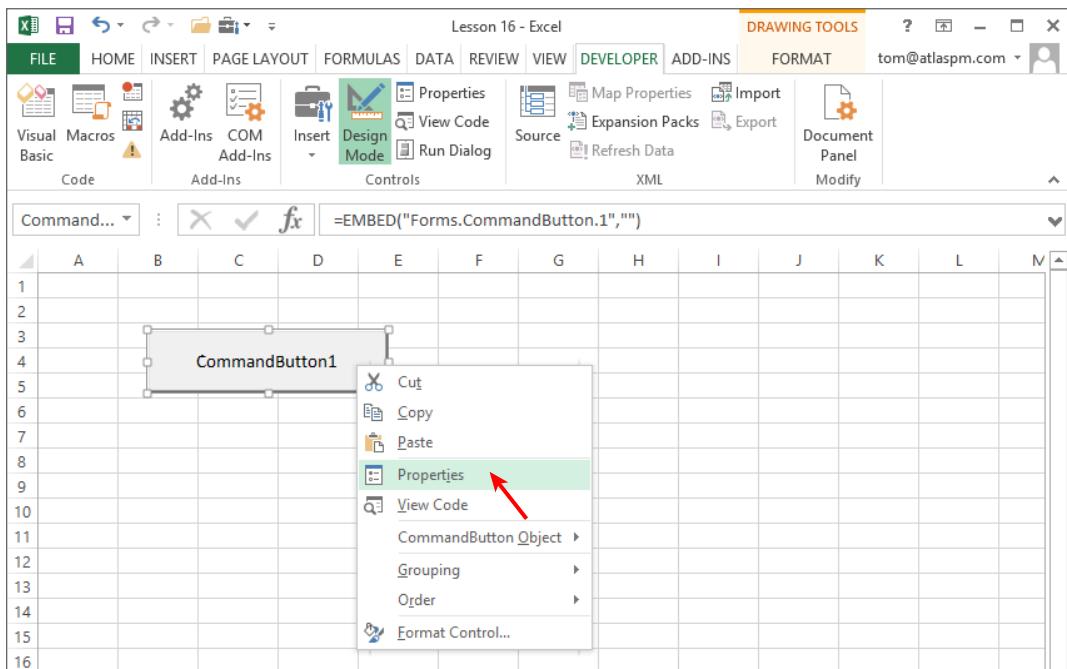


FIGURE 16-12

You will see the Properties window for the CommandButton, where you can modify a number of properties. Change the Caption property of the CommandButton to **CheckBox Checker**, as shown in Figure 16-13.

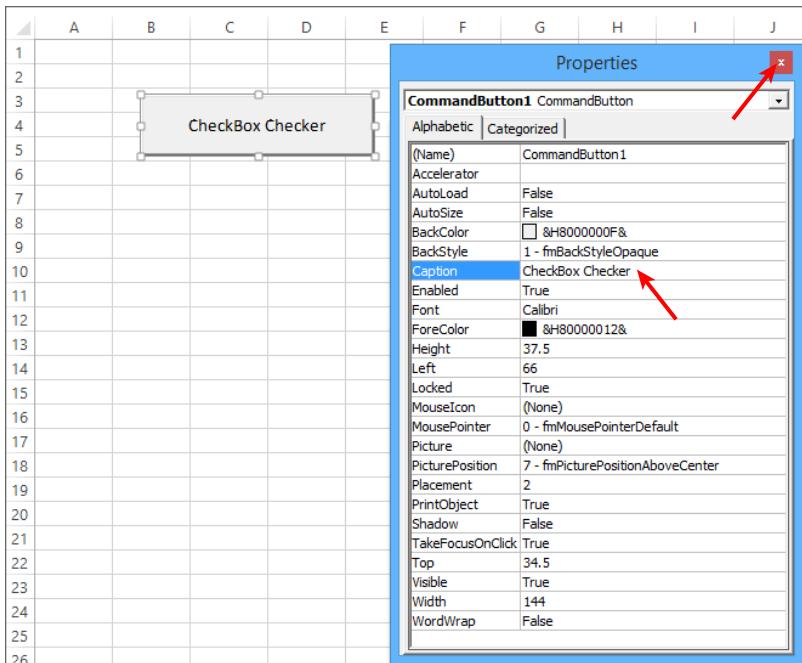


FIGURE 16-13

Draw a Label control and four Check Boxes from the Control Toolbox below the CommandButton. In Figure 16-14, I changed the Label's caption to Check Your Favorite Activities. I changed each CheckBox's caption to a different leisure activity.

Either double-click the CommandButton, or right-click it and select View Code. Either way, you go to the worksheet module and the default Click event is started for you with the following entry:

```
Private Sub CommandButton1_Click()
    End Sub
```

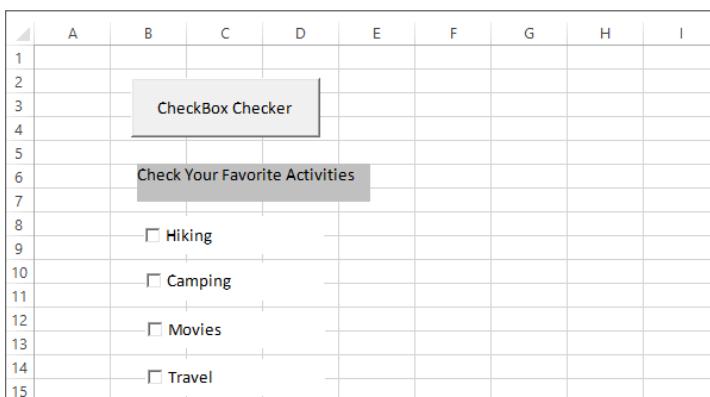


FIGURE 16-14

NOTE VBA code for embedded ActiveX objects is almost always in the module of the worksheet upon which the objects are embedded.

For this demonstration, when the CommandButton is clicked, it evaluates every embedded object on the worksheet. When the code comes across an ActiveX CheckBox, it determines whether the CheckBox is checked. At the end of the procedure, a message box appears, confirming how many (if any) CheckBoxes were checked, and their captions. The entire code looks as follows:

```

Private Sub CommandButton1_Click()
    'Evaluate which checkboxes are checked.

    'Declare an Integer type variable to help
    'count through the Checkboxes,'and an Object
    'type variable to identify the kind of ActiveX control
    '(checkboxes in this example) that are selected.
    Dim intCounter As Integer, xObj As OLEObject
    'Declare a String variable to list the captions
    'of selected checkboxes.
    Dim strObj As String

    'Start the Integer and String variables.
    intCounter = 0
    strObj = ""

    For Each xObj In ActiveSheet.OLEObjects
        If TypeName(xObj.Object) = "CheckBox" Then

            If xObj.Object.Value = True Then
                intCounter = intCounter + 1
                strObj = strObj & xObj.Object.Caption & Chr(10)
            End If

        End If
    Next xObj

    'Advise the user of your findings.
    If intCounter = 0 Then
        MsgBox "No CheckBoxes were selected.", , "Try to get out more often!"
    Else
        MsgBox "You selected " & intCounter & " CheckBox(es):" & vbCrLf & vbCrLf & _
        strObj, , "Here is what you checked:"
    End If

End Sub

```

Leave the VBE and return to the worksheet by pressing Alt+Q. Click the Design Mode button to exit Design Mode. Figure 16-15 shows where the Design Mode icon is on the Developer tab.

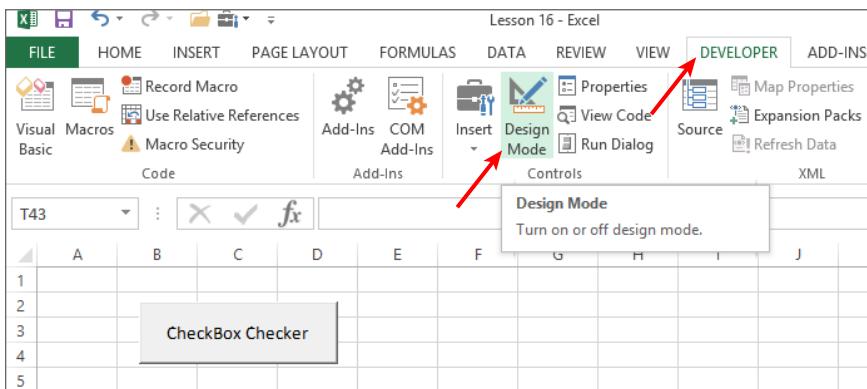


FIGURE 16-15

With Design Mode now off, you can test the `Click` event code for the ActiveX CommandButton. Figure 16-16 shows an example of the confirming message box when you click the CommandButton.

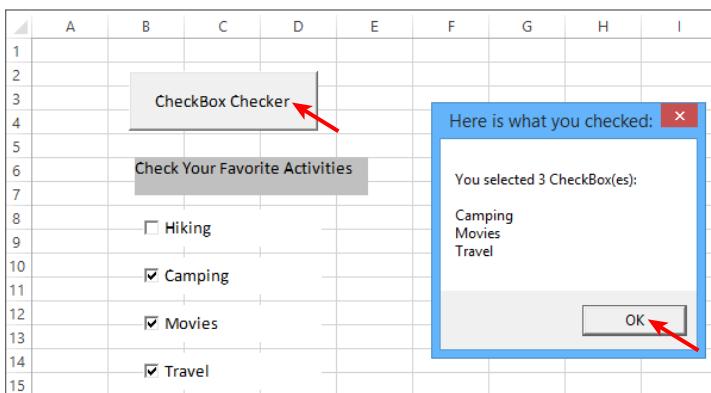


FIGURE 16-16

TRY IT

For this lesson, you place a Form Button on a worksheet that contains a hypothetical table of monthly income activity for a department store's clothing items. You attach a macro to the Button that, when clicked, toggles columns or rows as being hidden or visible, depending on how you want

to see the data. Upon each click of the Button, the cycle of views will be to see the entire table's detail, see totals only by clothing item, or see totals only by month. This lesson also includes tips on fast data entry by using the fill handle and shortcut keys.

Lesson Requirements

To get the sample workbook you can download Lesson 16 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open Excel and open a new workbook.
2. On your active worksheet, list the months of the year in range A6:A17. You can do this quickly by entering **January** in cell A6, then selecting A6, and pointing your mouse over the fill handle, which is the small black square in the lower-right corner of the selected cell. You know your mouse is hovering over the fill handle when the cursor changes to a crosshairs, as indicated in Figure 16-17. Press your left mouse button onto the fill handle, and drag your mouse down to cell A17 as indicated in Figure 16-18. Release the mouse button, and the 12 months of the year fill into range A6:A17 as shown in Figure 16-19.

A	B
1	
2	
3	
4	
5	
6	January
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	
19	

FIGURE 16-17

A	B
1	
2	
3	
4	
5	
6	January
7	
8	
9	
10	
11	
12	
13	
14	
15	
16	
17	
18	December
19	

FIGURE 16-18

A	B
1	
2	
3	
4	
5	
6	January
7	February
8	March
9	April
10	May
11	June
12	July
13	August
14	September
15	October
16	November
17	December
18	
19	

FIGURE 16-19

3. Enter some clothing items into range B5:F5.

4. Enter sample numbers in range B6:F17. There is nothing special about the numbers; they are just for demonstration purposes. To enter the numbers quickly, as shown in Figure 16-20, do the following:
- Select range B6:F17.
 - Type the formula `=INT(RAND()*1000)`.
 - Press Ctrl+Enter.
 - Press Ctrl+C to copy the range.
 - Right-click somewhere in the range B6:F17, and select Paste Special ➔ Values ➔ OK.
 - Press the Esc key to exit Copy mode.
5. In cell G5 enter **Total** and in cell A18 enter **Total**.
6. Select the column A header, which selects all of column A. Right-click any cell in column A, select Column Width, enter **20**, and click OK.
7. Quickly enter Sum functions for all rows and columns. Select range B6:G18, as shown in Figure 16-21, and either double-click the Sum function icon or press Alt+=.
8. With range B6:G18 currently selected, right-click anywhere in the selection, select Format Cells, and click the Number tab in the Format Cells dialog box. In the category pane select Currency, set Decimal Places to 0, and click OK as indicated in Figure 16-22. Your final result should resemble Figure 16-23, with different numbers because they were produced with the RAND function, but all good enough for this lesson.

	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5		Jackets	Shoes	Hats	Dresses	Shirts		
6	January	=INT(RAND()*1000)						
7	February							
8	March							
9	April							
10	May							
11	June							
12	July							
13	August							
14	September							
15	October							
16	November							
17	December							
18								
19								

FIGURE 16-20

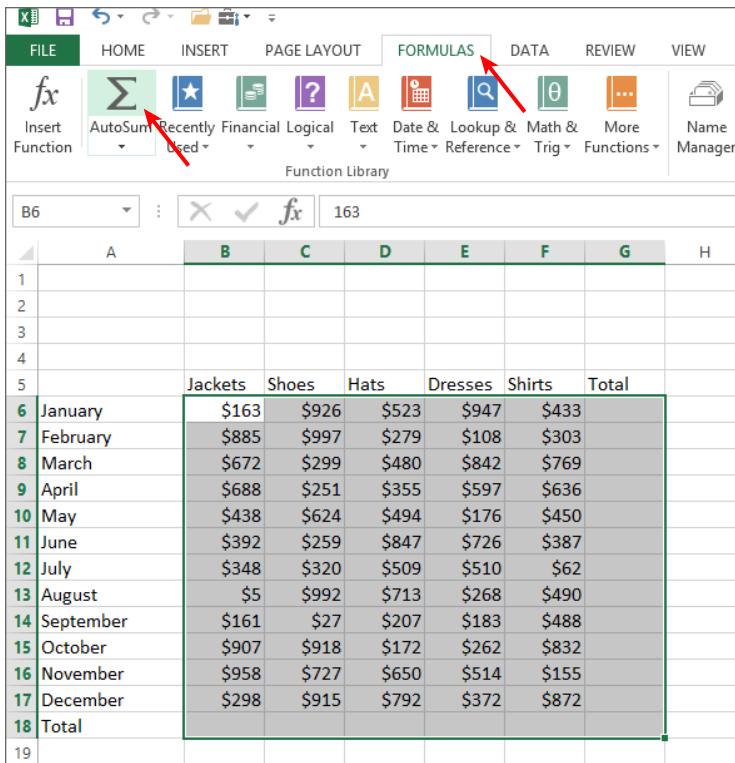


FIGURE 16-21

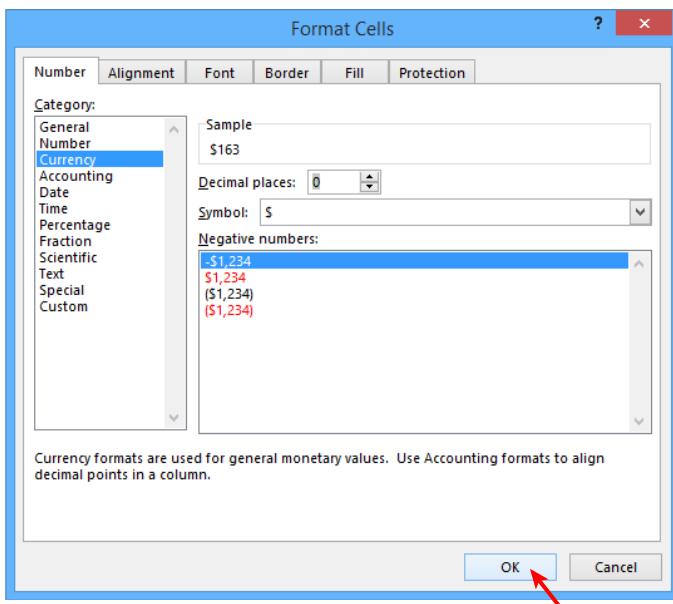


FIGURE 16-22

	A	B	C	D	E	F	G	H
1								
2								
3								
4								
5		Jackets	Shoes	Hats	Dresses	Shirts	Total	
6	January	\$163	\$926	\$523	\$947	\$433	\$2,992	
7	February	\$885	\$997	\$279	\$108	\$303	\$2,572	
8	March	\$672	\$299	\$480	\$842	\$769	\$3,062	
9	April	\$688	\$251	\$355	\$597	\$636	\$2,527	
10	May	\$438	\$624	\$494	\$176	\$450	\$2,182	
11	June	\$392	\$259	\$847	\$726	\$387	\$2,611	
12	July	\$348	\$320	\$509	\$510	\$62	\$1,749	
13	August	\$5	\$992	\$713	\$268	\$490	\$2,468	
14	September	\$161	\$27	\$207	\$183	\$488	\$1,066	
15	October	\$907	\$918	\$172	\$262	\$832	\$3,091	
16	November	\$958	\$727	\$650	\$514	\$155	\$3,004	
17	December	\$298	\$915	\$792	\$372	\$872	\$3,249	
18	Total	\$5,915	\$7,255	\$6,021	\$5,505	\$5,877	\$30,573	
19								

FIGURE 16-23

9. The task at hand is to create a macro that will be attached to a Form Button. Each time you click the Button, the macro toggles to the next of three different views of the table: seeing the entire table's detail, seeing totals only by clothing item, or seeing totals only by month. To get started, press Alt+F11 to go to the Visual Basic Editor.

10. From the VBE menu bar, click Insert \Rightarrow Module.

11. In your new module, type **Sub ToggleViews** and press Enter. VBA produces the following two lines of code, with an empty row between them:

```
Sub ToggleViews()
```

```
End Sub
```

12. Because the macro hides and unhides rows and columns, turn off **ScreenUpdating** to keep the screen from flickering:

```
Application.ScreenUpdating = False
```

13. Open a **With** structure that uses **Application.Caller** to identify the Form Button that was clicked:

```
With ActiveSheet.Buttons(Application.Caller)
```

14. Toggle between views based on the Button's captions to determine which view is next in the cycle:

```
If .Caption = "SHOW ALL" Then
With Range("A5:G18")
.EntireColumn.Hidden = False
.EntireRow.Hidden = False
End With
.Caption = "MONTH TOTALS"
```

```
ElseIf .Caption = "MONTH TOTALS" Then
    Range("B:F").EntireColumn.Hidden = True
    .Caption = "ITEM TOTALS"
ElseIf .Caption = "ITEM TOTALS" Then
    Range("B:F").EntireColumn.Hidden = False
    Rows("6:17").Hidden = True
    .Caption = "SHOW ALL"
End If 'for evaluating the button caption.
```

15. Close the With structure for Application.Caller:

```
End With
```

16. Turn ScreenUpdating on again:

```
Application.ScreenUpdating = True
```

17. Your entire macro looks like this:

```
Sub ToggleViews()

    'Turn off ScreenUpdating.
    Application.ScreenUpdating = False

    'Open a With structure that uses Application.Caller
    'to identify the Form Button that was clicked.
    With ActiveSheet.Buttons(Application.Caller)

        'Toggle between views based on the Button's captions
        'to determine which view is next in the cycle.
        If .Caption = "SHOW ALL" Then
            With Range("A5:G18")
                .EntireColumn.Hidden = False
                .EntireRow.Hidden = False
            End With
            .Caption = "MONTH TOTALS"

        ElseIf .Caption = "MONTH TOTALS" Then
            Range("B:F").EntireColumn.Hidden = True
            .Caption = "ITEM TOTALS"

        ElseIf .Caption = "ITEM TOTALS" Then
            Range("B:F").EntireColumn.Hidden = False
            Rows("6:17").Hidden = True
            .Caption = "SHOW ALL"

        End If 'for evaluating the Button caption.

        'Close the With structure for Application.Caller.
    End With

    'Turn ScreenUpdating on again.
    Application.ScreenUpdating = True

End Sub
```

18. Press Alt+Q to return to the worksheet.
19. Draw a Form Button on your worksheet at the top of column A. When you release the mouse button you see the Assign Macro dialog box. Select the macro named `ToggleViews` and click OK, as shown in Figure 16-24.

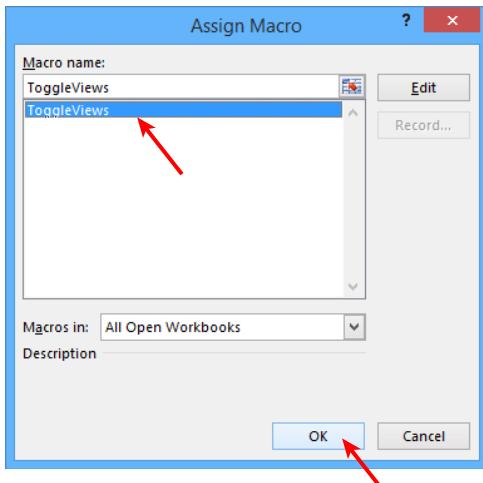


FIGURE 16-24

20. Make sure the Button is totally within column A, as indicated in Figure 16-25. Right-click the Button and select Edit Text.

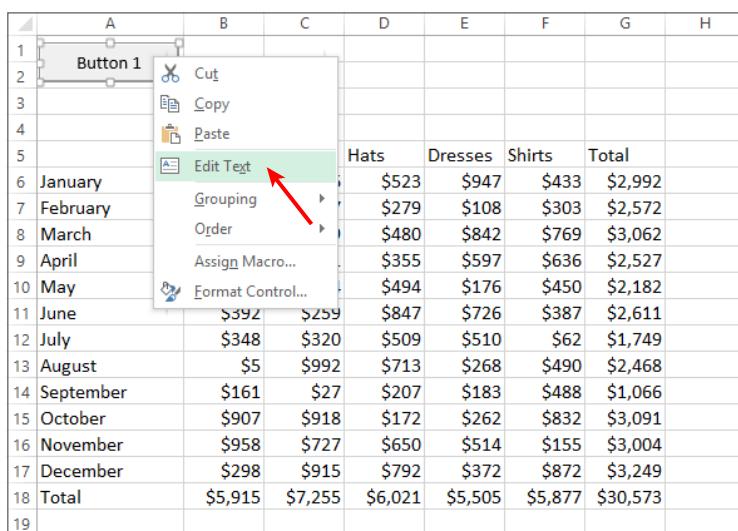


FIGURE 16-25

21. Change the Button's caption to `SHOW ALL`, as shown in Figure 16-26.

A	B	C	D	E	F	G	H
1	SHOW ALL						
2							
3							
4							
5	Jackets	Shoes	Hats	Dresses	Shirts	Total	
6	January	\$163	\$926	\$523	\$947	\$433	\$2,992
7	February	\$885	\$997	\$279	\$108	\$303	\$2,572
8	March	\$672	\$299	\$480	\$842	\$769	\$3,062
9	April	\$688	\$251	\$355	\$597	\$636	\$2,527
10	May	\$438	\$624	\$494	\$176	\$450	\$2,182
11	June	\$392	\$259	\$847	\$726	\$387	\$2,611
12	July	\$348	\$320	\$509	\$510	\$62	\$1,749
13	August	\$5	\$992	\$713	\$268	\$490	\$2,468
14	September	\$161	\$27	\$207	\$183	\$488	\$1,066
15	October	\$907	\$918	\$172	\$262	\$832	\$3,091
16	November	\$958	\$727	\$650	\$514	\$155	\$3,004
17	December	\$298	\$915	\$792	\$372	\$872	\$3,249
18	Total	\$5,915	\$7,255	\$6,021	\$5,505	\$5,877	\$30,573
19							

FIGURE 16-26

22. Select any cell to deselect the Button. Click the Button once and nothing changes on the sheet because all the columns and rows are already visible. You see that the Button's caption changed to MONTH TOTALS. If you click the Button again, you see the month names listed in column A, and their totals listed in column G. The Button's caption reads ITEM TOTALS. Click the Button again to see the clothing items named in row 5, and their totals listed in row 18. The Button's caption reads SHOW ALL, and if you click the Button again, all rows and columns are shown.
23. You can continue cycling through the table's views by clicking the Form Button for each view that you coded into the `ToggleViews` macro.

REFERENCE Please select the video for Lesson 16 at www.wrox.com/go/excelvba24hour. You will also be to download the code and resources for this lesson from the website.

17

Programming Charts

When I started to program Excel in the early 1990s, I remember being impressed with the charting tools that came with Excel. They were very good back then, and today's chart features in Excel are downright awesome, rivaling—and usually surpassing—the charting packages of any software application.

Because you are reading this book, chances are pretty good that you've manually created your share of charts in Excel using the Chart Wizard or by selecting a chart type from the dozens of choices on the Ribbon. You might also have played with the Macro Recorder to do some automation of chart creation. This lesson takes you past the Macro Recorder's capabilities to show how to create and manipulate embedded charts and chart sheets.

The topic of charting is one that can, and does, fill entire books. The myriad chart types and features that Excel makes available to you goes well beyond the scope of this lesson. What this lesson does is show you the syntaxes for several methods that work for embedded charts and chart sheets, with a few different features and chart types represented in the programming code. From the VBA examples in this lesson, you can expand your chart programming skills by substituting the chart types and features shown for others that may be more suited to the kinds of charts you want to develop.

NOTE *In the examples, you might notice that the charts being created are declared as a Chart type object variable, which makes it easier to refer to the charts when you want to manipulate them in code. In any case, Excel has two separate object models for charts. For a chart on its own chart sheet, it is a Chart object. For a chart embedded on a worksheet, it is a ChartObject object. Chart sheets are members of the workbook's Charts collection, and each ChartObject on a worksheet is a member of the worksheet's ChartObjects collection.*

ADDING A CHART TO A CHART SHEET

As you know, a chart sheet is a special kind of sheet in your workbook that contains only a chart. If the chart is destined to be large and complicated, users often prefer such a chart be on its own sheet so they can view its detail more easily.

Figure 17-1 shows a table of sales by month for a company that is the source data for this chart example. The table is on Sheet1, and although you can correctly refer to the source range in your code as A1:B13, I prefer using the `CurrentRegion` property to reduce the chances of entering the wrong range reference in my code.

The following macro creates a column chart for a new chart sheet based on the data in Figure 17-1. If the `Location` property of your `Chart` object has not been specified, as it has not been in this macro, your chart is created in its own chart sheet. The result of this new chart sheet is shown in Figure 17-2.

```
Sub CreateChartSheet()
    'Declare your chart type object variable.
    Dim myChart1 As Chart
    'Set your variable to add a chart.
    Set myChart1 = Charts.Add
    'Define the new chart's source data.
    myChart1.SetSourceData _
        Source:=Worksheets("Sheet1").Range("A1").CurrentRegion, _
        PlotBy:=xlColumns
    'Define the type of chart.
    myChart1.ChartType = xlColumnClustered
    'Delete the legend because it is redundant with the chart title.
    ActiveChart.Legend.Delete
End Sub
```

	A	B	C
1	Month	Sales	
2	January	71605	
3	February	73632	
4	March	90114	
5	April	87041	
6	May	63362	
7	June	73417	
8	July	46648	
9	August	14292	
10	September	62041	
11	October	65849	
12	November	37370	
13	December	73112	
14			

FIGURE 17-1

NOTE To change your default type of chart, right-click any chart in your workbook and select *Change Chart Type*. In the *Change Chart Type* dialog box, select a chart type, click the *Set as Default Chart* button, and click *OK*. In version 2013, right-click a chart type and select *Set as Default Chart*.

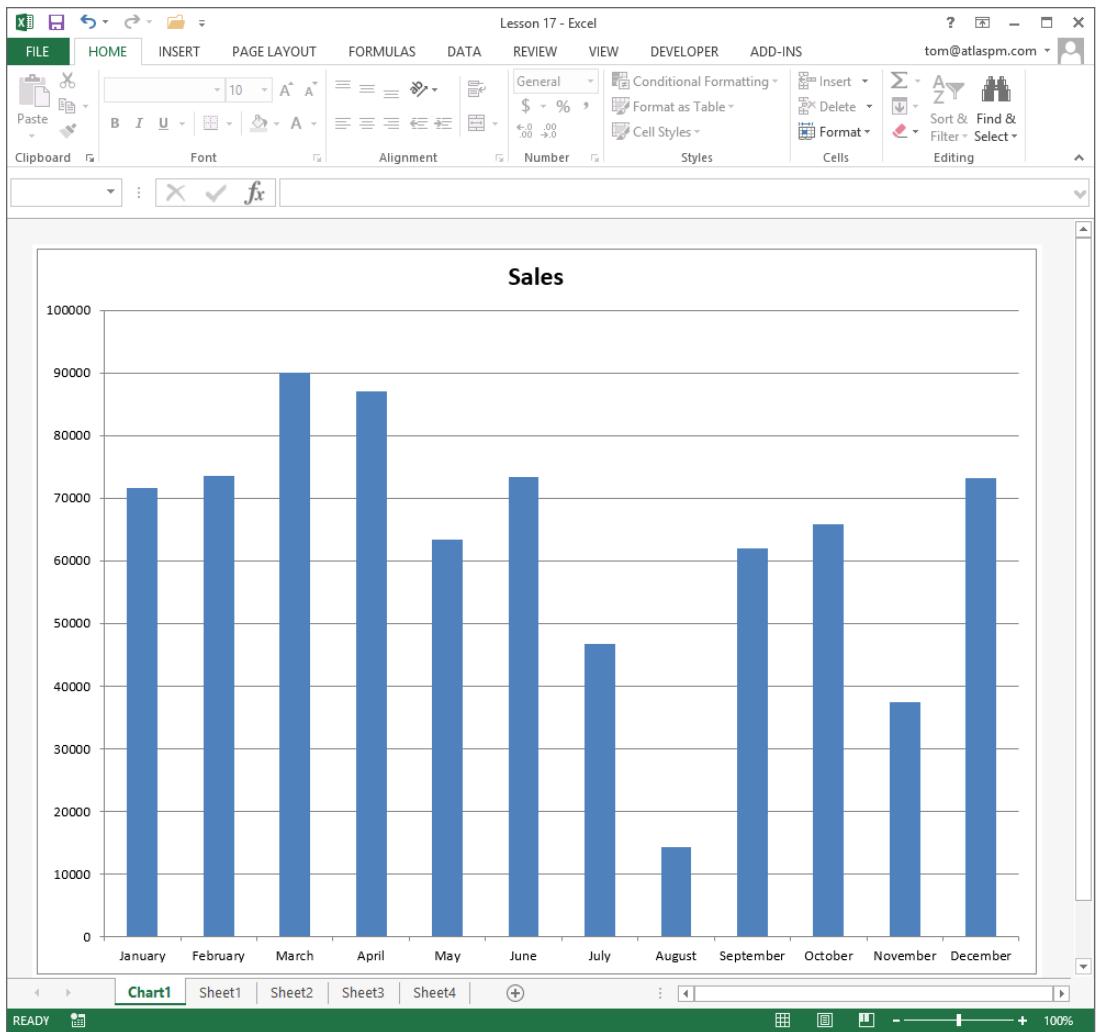


FIGURE 17-2

Simply executing the code line `Charts.Add` in the Immediate window creates a new chart sheet. If the active cell were within a table of data, your default type chart would occupy the new chart sheet, representing the table, or more precisely, the data within the `CurrentRegion` property of the selected cell. If you did not have any data selected at the time, a new chart sheet would still be created, with a blank `Chart` object looking like an empty canvas waiting to be supplied with source data.

DID YOU KNOW...

If the active cell is within a table of data, or you have a range of data selected, and you press the F11 key, a new chart sheet is added to hold a chart that represents the selected data. Some people find this to be an annoyance because they have no interest in charts, and may not be aware they touched the F11 key when a chart sheet has appeared out of nowhere.

If you want to negate the effect of pressing the F11 key, you can place the following OnKey procedures into the ThisWorkbook module. Some Excel users who frequently use the F2 key to get into Edit mode sometimes press the F1 Help key by mistake and nullify the F1 key in this fashion as well:

```
Private Sub Workbook_Open()
Application.OnKey "{F11}", ""
End Sub

Private Sub Workbook_Activate()
Application.OnKey "{F11}", ""
End Sub

Private Sub Workbook_Deactivate()
Application.OnKey "{F11}"
End Sub

Private Sub Workbook_BeforeClose(Cancel As Boolean)
Application.OnKey "{F11}"
End Sub
```

ADDING AN EMBEDDED CHART TO A WORKSHEET

When you embed a chart in a worksheet, there is more to consider than when you create a chart for its own chart sheet. When you embed a chart, you need to specify which worksheet you want the chart to be on (handled by the Location property), and where on the worksheet you want the chart to be placed. The following macro is an example of how to place a column chart into range D3:J20 of the active worksheet, close to the source range, as shown in Figure 17-3:

```
Sub CreateChartSameSheet()
'Declare an Object variable for the chart
'and for the embedded ChartObject.
Dim myChart1 As Chart, cht1 As ChartObject
'Declare a Range variable to specify what range
'the chart will occupy, and on what worksheet.
Dim rngChart1 As Range, DestinationSheet As String

'The chart will be placed on the active worksheet.
DestinationSheet = ActiveSheet.Name

'Add a new chart
```

```

Set myChart1 = Charts.Add

'Specify the chart's location as the active worksheet.
Set myChart1 =
myChart1.Location =
(Where:=xlLocationAsObject, Name:=DestinationSheet)

'Define the new chart's source data
myChart1.SetSourceData =
Source:=Range("A1").CurrentRegion, PlotBy:=xlColumns

'Define the type of chart, in this case, a Column chart.
myChart1.ChartType = xlColumnClustered

'Activate the chart to identify its ChartObject.
'The (1) assumes this is the first (index #1) chart object
'on the worksheet.
ActiveSheet.ChartObjects(1).Activate
Set cht1 = ActiveChart.Parent

'Specify the range you want the chart to occupy.
Set rngChart1 = Range("D3:J20")
cht1.Left = rngChart1.Left
cht1.Width = rngChart1.Width
cht1.Top = rngChart1.Top
cht1.Height = rngChart1.Height

'Deselect the chart by selecting a cell.
Range("A1").Select
End Sub

```

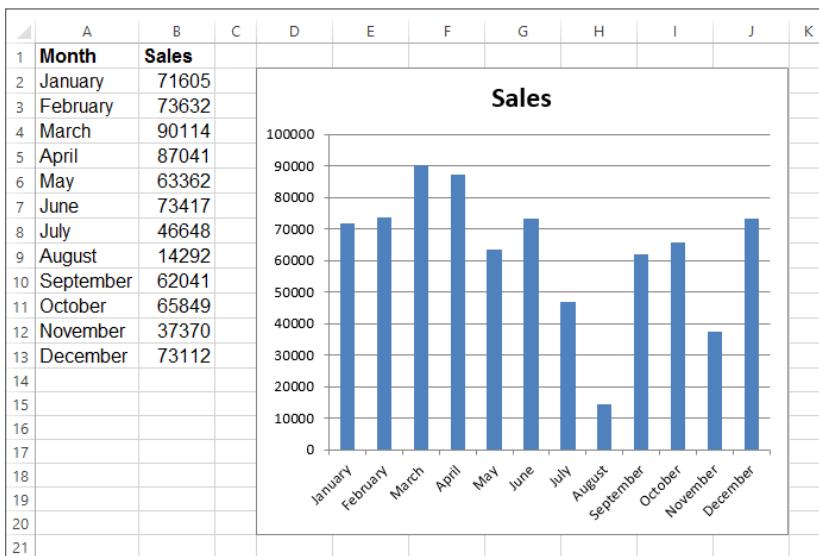


FIGURE 17-3

WARNING Here's a cool tip: Starting with version 2010, you can select any cell in a table of data, then press Alt+F1 to embed a chart of that data onto your worksheet. From there, you can drag the chart to your preferred location on the worksheet.

NOTE One of the best practice items in VBA programming that I mention throughout the book, and you will see posted in newsgroups ad nauseam, is to avoid selecting or activating objects in your VBA code. Most of the time that is good advice. However, sometimes you need to select objects to refer reliably to them or to manipulate them, and the preceding macro demonstrated two examples. The ChartObject was activated to derive the actual name of the chart. Also, the macro ended with cell A1 being selected. You could select any cell or any object, but a cell—any cell—is the safest object to select after creating a new embedded chart. Any code that is executed after adding a new chart might not execute correctly if the ChartObject is still selected. The most reliable way to deselect a chart at the end of your macro is to select a cell.

MOVING A CHART

You can change the location of any chart, which you might be familiar with if you've right-clicked a chart's area and noticed the Move Chart menu item. The following scenarios show how to do this with VBA.

To move a chart from a chart sheet to a worksheet, select the chart sheet programmatically and specify the worksheet where you want the chart to be relocated. It's usually a good idea to tell VBA where on the worksheet you want the chart to go; otherwise, the chart is plopped down on the sheet wherever VBA decides. That is why the code in the `With` structure specifies that cell C3 be the top-left corner of the relocated chart:

```
Sub ChartSheetToWorksheet()
    'Chart1 is the name of the chart sheet.
    Sheets("Chart1").Select
    'Move the chart to Sheet1.
    ActiveChart.Location Where:=xlLocationAsObject, Name:="Sheet1"

    'Cell C3 is the top left corner location of the chart.
    With Worksheets("Sheet1")
        ActiveChart.Parent.Left = .Range("C3").Left
        ActiveChart.Parent.Top = .Range("C3").Top
    End With

    'Deselect the chart.
    Range("A1").Select

End Sub
```

To move a chart from a worksheet to a chart sheet, you need to determine the name or index number of your chart. If you have only one chart on your worksheet, you know that chart's index property is 1, but specifying the chart by its name is a safe way to go. The code is much simpler because a chart sheet can contain only one chart, so you don't need to specify a location on the chart sheet itself:

```
Sub EmbeddedChartToChartSheet()
    ActiveSheet.ChartObjects("Chart 1").Activate
    ActiveChart.Location Where:=xlLocationAsNewSheet, Name:="Chart1"
End Sub
```

NOTE You can determine the name of any embedded chart quickly by selecting it to see its name in the Name box.

To move an embedded chart from one worksheet to another, it's the same concept of specifying which chart to move, and which worksheet to move it to:

```
Sub EmbeddedChartToAnotherWorksheet()
    'Chart 5 is the name of the chart to move to Sheet2.
    ActiveSheet.ChartObjects("Chart 5").Activate
    ActiveChart.Location Where:=xlLocationAsObject, Name:="Sheet2"

    'Cell B6 is the top left corner location of the chart.
    With Worksheets("Sheet2")
        ActiveChart.Parent.Left = .Range("B6").Left
        ActiveChart.Parent.Top = .Range("B6").Top
    End With

    'Deselect the chart.
    Range("A1").Select

End Sub
```

You can quickly move all chart sheets to their own workbook. For example, check out the following example that creates a new workbook and relocates the chart sheets before Sheet1 in that new workbook:

```
Sub ChartSheetsToWorkbook()
    'Declare variable for your active workbook name.
    Dim myName As String
    'Define the name of your workbook.
    myName = ActiveWorkbook.Name
    'Add a new Excel workbook.
    Workbooks.Add 1
    'Copy the chart sheets from your source workbook
    'to the new workbook.
    Workbooks(myName).Charts.Move before:=Sheets(1)
End Sub
```

LOOPING THROUGH ALL EMBEDDED CHARTS

Suppose you want to do something to every embedded chart in your workbook. For example, if some charts were originally created with different background colors, you might want to standardize the look of all charts to have the same color scheme. The following macro shows how to loop through every chart on every worksheet to format the chart area with a standard color of cyan:

```
Sub LoopAllEmbeddedCharts()

    'Turn off ScreenUpdating.
    Application.ScreenUpdating = False

    'Declare variables for worksheet and chart objects.
    Dim wks As Worksheet, ChObj As ChartObject

    'Open loop for every worksheet.
    For Each wks In Worksheets

        'Determine if the worksheet has at least one chart.
        If wks.ChartObjects.Count > 0 Then

            'If the worksheet has a chart, activate the worksheet.
            wks.Activate

            'Loop through each chart object.
            For Each ChObj In ActiveSheet.ChartObjects

                'Activate the chart.
                ChObjActivate

                'Color the chart area cyan.
                ActiveChart.ChartArea.Interior.ColorIndex = 8

                'Deselect the active chart before proceeding to the
                'next chart or the next worksheet.
                Range("A1").Select

                'Continue and close the loop for every chart on that sheet.
                Next ChObj

            'Close the If structure if the worksheet had no chart.
            End If

            'Continue and close the loop for every worksheet.
            Next wks

        'Turn on ScreenUpdating.
        Application.ScreenUpdating = True

    End Sub
```

If you have chart sheets to be looped through, the code must be different to take into account the type of sheet to look for, because a chart sheet is a different type of sheet than a worksheet. This macro accomplishes the same task of coloring the chart area, but for charts on chart sheets:

```
Sub LoopAllChartSheets()

    'Turn off ScreenUpdating.
    Application.ScreenUpdating = False

    'Declare an object variable for the Sheets collection.
    Dim objSheet As Object

    'Loop through all sheets, only looking for a chart sheet.
    For Each objSheet In ActiveWorkbook.Sheets
        If TypeOf objSheet Is Excel.Chart Then

            'Activate the chart sheet.
            objSheet.Activate

            'Color the chart area cyan.
            ActiveChart.ChartArea.Interior.ColorIndex = 8

            'Close the If structure and move on to the next sheet.
            End If
        Next objSheet

        'Turn on ScreenUpdating.
        Application.ScreenUpdating = True

    End Sub
```

Deleting Charts

To delete all charts on a worksheet, you can execute this code line in the Immediate window, or as part of a macro:

```
If activesheet.ChartObjects.Count > 0 Then ActiveSheet.ChartObjects.Delete
```

To delete chart sheets, loop through each sheet starting with the last sheet, determine whether the sheet is a chart sheet, and if so, delete it.

NOTE *This loop starts from the last sheet and moves backward using the Step -1 statement. It's a wise practice to loop backward when deleting sheets, rows, or columns. Behind the scenes, VBA relies on the counts of objects in collections, and where the objects are located relative to the others. Deleting objects starting at the end and working your way to the beginning keeps VBA's management of those objects in order.*

```
Sub DeleteChartSheets()

    'Turn off ScreenUpdating. Also turn off the Alerts feature,
    'so when you delete a sheet VBA does not warn you.
    With Application
        .ScreenUpdating = False
        .DisplayAlerts = False

        'Declare an Integer variable for the count of all Sheets.
        Dim intSheet As Integer
        'Loop through all sheets, only looking for a chart sheet.
        For intSheet = Sheets.Count To 1 Step -1
            If TypeName(Sheets(intSheet)) = "Chart" Then Sheets(intSheet).Delete
        Next intSheet

        'Turn on ScreenUpdating and DisplayAlerts.
        .DisplayAlerts = True
        .ScreenUpdating = True
    End With

End Sub
```

Renaming a Chart

As you have surely noticed when creating objects such as charts, pivot tables, or drawing objects, Excel has a refined knack for giving those objects the blandest default names imaginable. Suppose you have three embedded charts on your worksheet. The following macro changes the names of those charts to something more meaningful:

```
Sub RenameCharts()
    With ActiveSheet
        .ChartObjects(1).Name = "Monthly Income"
        .ChartObjects(2).Name = "Monthly Expense"
        .ChartObjects(3).Name = "Net Profit"
    End With
End Sub
```

TRY IT

In this lesson you create an embedded pie chart, position it near the source data, and give each legend key a unique color. The pie has four slices; each has a unique color and displays its respective data label.

Lesson Requirements

To get the sample database files you can download Lesson 17 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Insert a new worksheet and construct the simple table, as shown in Figure 17-4.

A	B	C	D
1	Quarterly Sales		
2			
3	Quarter	Sales	Percent
4	Quarter 1	\$591,254	28%
5	Quarter 2	\$326,589	15%
6	Quarter 3	\$486,234	23%
7	Quarter 4	\$745,698	35%
8	Total	\$2,149,775	
9			

FIGURE 17-4

2. From your worksheet, press Alt+F11 to go to the Visual Basic Editor.
3. From the VBE menu bar, click Insert \leftrightarrow Module.
4. In your new module, enter the name of this macro, which I am calling TryItPieChart. Type `Sub TryItPieChart`, press Enter, and VBA produces the following code:

```
Sub TryItPieChart()
```

```
End Sub
```

5. Declare the ChartObject variable:

```
Dim chtQuarters As ChartObject
```

6. Set the variable to the chart being added. Position the chart near the source data:

```
Set chtQuarters =  
    ActiveSheet.ChartObjects.Add  
    (Left:=240, Width:=340, Top:=5, Height:=240)
```

NOTE The data components inside the parentheses tell VBA where to position your new chart on the worksheet.

The `Left` parameter defines the position in points of the left edge of the `ChartObject` relative to the left edge of the worksheet.

The `Top` parameter defines the position in points of the top of the `ChartObject` relative to the top of the worksheet.

The `Width` parameter defines the `ChartObject`'s width, in points.

The `Height` parameter defines the `ChartObject`'s height, in points.

A *point* is a small unit of measurement (an inch is approximately 72 points).

7. Define the range for this pie chart:

```
chtQuarters.Chart.SetSourceData Source:=Range("A3:B7")
```

8. Define the type of chart, which is a pie:

```
chtQuarters.Chart.ChartType = xlPie
```

9. Activate the new chart to work with it:

```
ActiveSheet.ChartObjects(1).Activate
```

10. Color the legend entries to identify each pie piece:

```
With ActiveChart.Legend  
    .LegendEntries(1).LegendKey.Interior.Color = vbYellow  
    .LegendEntries(2).LegendKey.Interior.Color = vbCyan  
    .LegendEntries(3).LegendKey.Interior.Color = vbRed  
    .LegendEntries(4).LegendKey.Interior.Color = vbGreen  
End With
```

11. Add data labels to see the numbers in the pie slices:

```
ActiveChart.SeriesCollection(1).ApplyDataLabels
```

12. Edit the chart title's text:

```
ActiveChart.ChartTitle.Text = "Quarterly Sales"
```

13. Format the legend:

```
ActiveChart.Legend.Select  
With Selection.Font  
    .Name = "Arial"  
    .FontStyle = "Bold"  
    .Size = 14  
End With
```

14. Deselect the chart by selecting a cell:

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

15. Press Alt+Q to return to the worksheet, and test your macro, which in its entirety looks like the following code. The result looks like Figure 17-5, with a pie chart positioned near the source data.

```
Sub TryItPieChart()  
  
    'Declare the ChartObject variable.  
    Dim chtQuarters As ChartObject  
  
    'Set the variable to the chart being added.  
    'Position the chart near the source data.  
    Set chtQuarters =  
        ActiveSheet.ChartObjects.Add _  
        (Left:=240, Width:=340, Top:=5, Height:=240)  
  
    'Define the range for this pie chart.  
    chtQuarters.Chart.SetSourceData Source:=Range("A3:B7")  
  
    'Define the type of chart, which is a pie.
```

```

chtQuarters.Chart.ChartType = xlPie

'Activate the new chart to work with it.
ActiveSheet.ChartObjects(1).Activate

'Color the legend entries to identify each pie piece.
With ActiveChart.Legend
    .LegendEntries(1).LegendKey.Interior.Color = vbYellow
    .LegendEntries(2).LegendKey.Interior.Color = vbCyan
    .LegendEntries(3).LegendKey.Interior.Color = vbRed
    .LegendEntries(4).LegendKey.Interior.Color = vbGreen
End With

'Add data labels to see the numbers in the pie slices.
ActiveChart.SeriesCollection(1).ApplyDataLabels

>Edit the chart's title text.
ActiveChart.ChartTitle.Text = "Quarterly Sales"

'Format the legend.
ActiveChart.Legend.Select
With Selection.Font
    .Name = "Arial"
    .FontStyle = "Bold"
    .Size = 14
End With

'Deselect the chart by selecting a cell.
Range("A1").Select

End Sub

```

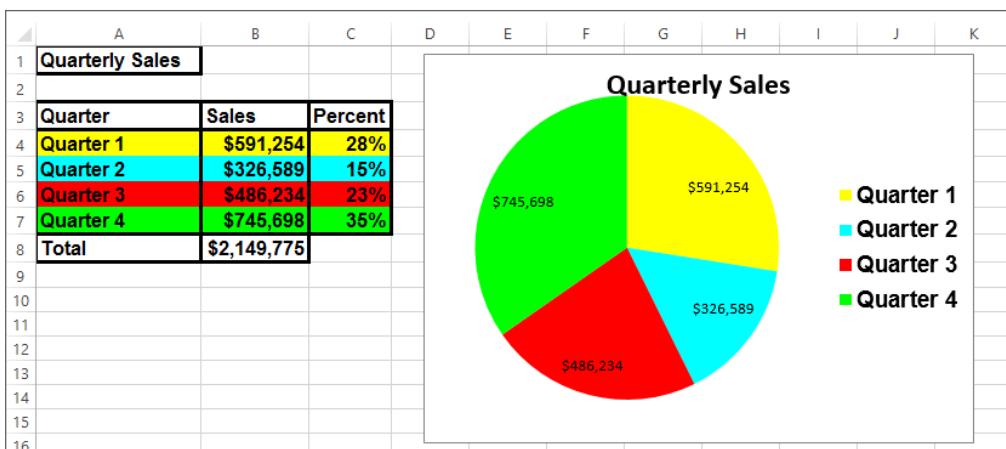


FIGURE 17-5

REFERENCE Please select the video for Lesson 17 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

18

Programming PivotTables and PivotCharts

PivotTables are Excel's most powerful feature. They are an amazing tool that can summarize more than a million rows of data into concise, meaningful reports in a matter of seconds. You can format the reports in many ways, and include an interactive chart to complement the reports at no extra cost of time.

If you are not familiar with PivotTables, you are not alone. Surveys of Excel users worldwide have consistently indicated that far less than half of those surveyed said they use PivotTables, including people who use Excel throughout their entire workday. Because PivotTables are worth becoming familiar with, this lesson starts with an overview of PivotTables and PivotCharts, followed by examples of how to create and manipulate them programmatically with VBA.

CREATING A PIVOTTABLE REPORT

Suppose you manage the clothing sales department for a national department store. You receive tens of thousands of sales records from your stores all over the country, with lists that look similar to Figure 18-1. With lists this large, it's impossible to gain any meaningful insight into trends or marketing opportunities unless you can organize the data in a summarized fashion.

If you select a single cell anywhere in the list, such as cell E7, which is selected in Figure 18-2, you can create a PivotTable by selecting the Insert tab and clicking the PivotTable icon. The Create PivotTable dialog box appears with the Table/Range field already filled in, as shown in Figure 18-3. I chose to keep the PivotTable on the same worksheet as the source data, and for the PivotTable's top-left corner to occupy cell H4.

	A	B	C	D	E	F	G
1	Store ID	Region	Item	When	Quantity	Revenue	
2	Store 9	West	Jackets	Quarter 1	1632	6045	
3	Store 3	North	Jackets	Quarter 2	1028	3808	
4	Store 7	West	Pants	Quarter 3	574	2127	
5	Store 6	South	Jackets	Quarter 4	2059	7628	
6	Store 6	South	Shirts	Quarter 1	217	804	
7	Store 8	West	Hats	Quarter 2	116	430	
8	Store 5	South	Jackets	Quarter 3	2179	8074	
9	Store 3	North	Pants	Quarter 4	150	558	
10	Store 1	East	Jackets	Quarter 1	1695	6281	
11	Store 5	South	Jackets	Quarter 2	1595	5908	
12	Store 5	South	Jackets	Quarter 3	2152	7972	
13	Store 5	South	Jackets	Quarter 4	822	3048	
14	Store 4	North	Shirts	Quarter 3	1217	4508	
15	Store 7	West	Shirts	Quarter 1	5555	6548	
16	Store 4	North	Hats	Quarter 3	1767	6548	
17	Store 2	East	Hats	Quarter 4	440	1632	
18	Store 4	North	Jackets	Quarter 1	1220	4521	
19	Store 5	South	Jackets	Quarter 2	1203	4457	
20	Store 9	West	Jackets	Quarter 3	1244	4609	
21	Store 5	South	Jackets	Quarter 4	1292	4788	
22	Store 3	North	Jackets	Quarter 1	927	3434	
23	Store 1	East	Hats	Quarter 2	2178	8067	
24	Store 5	South	Hats	Quarter 3	1563	5791	
25	Store 5	South	Jackets	Quarter 4	2512	9307	
26	Store 1	East	Jackets	Quarter 1	887	336	
27	Store 3	North	Hats	Quarter 2	9987	2020	
28	Store 8	West	Shirts	Quarter 3	2081	7711	
29	Store 4	North	Shirts	Quarter 4	61	229	
30	Store 2	East	Scarves	Quarter 1	2617	9694	
31	Store 6	South	Jackets	Quarter 2	660	2447	
32	Store 6	South	Jackets	Quarter 3	2529	9367	
33	Store 8	West	Jackets	Quarter 4	586	2172	
34	Store 7	West	Jackets	Quarter 1	683	2530	
35	Store 4	North	Pants	Quarter 2	1775	6577	
36	Store 2	East	Hats	Quarter 3	953	3531	

FIGURE 18-1

NOTE When placing a PivotTable on the same worksheet alongside the source table, it's best to have at least one empty column between the source table and your PivotTable. It's also a good idea to leave a few empty rows above the PivotTable to make room for the Filters area (what was called the Page area in Excel version 2003).

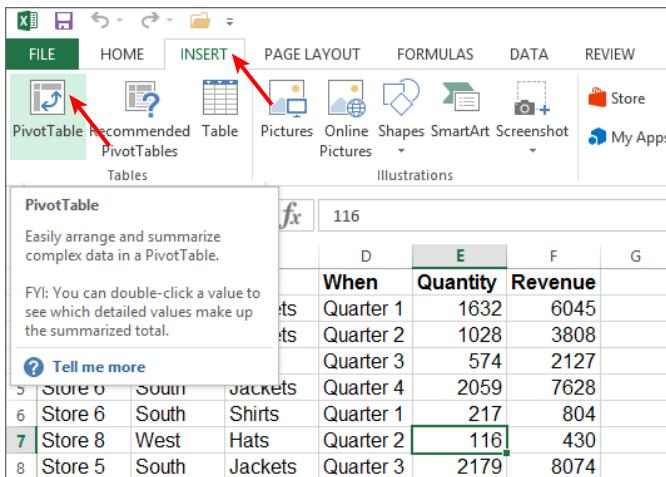


FIGURE 18-2

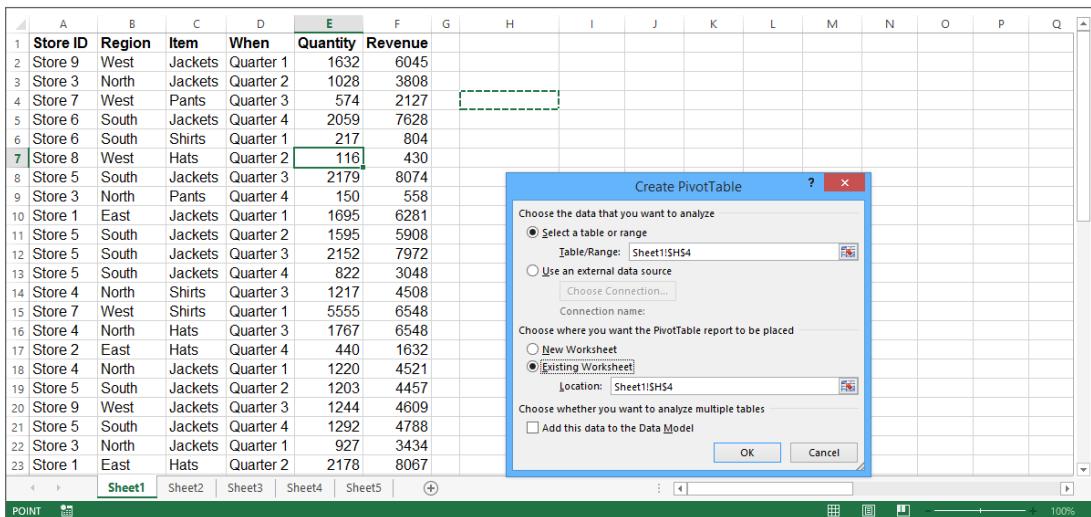


FIGURE 18-3

Using Excel version 2013, when you click OK you see an image similar to Figure 18-4, with the representation of where the PivotTable will be, and the Field List at the right.

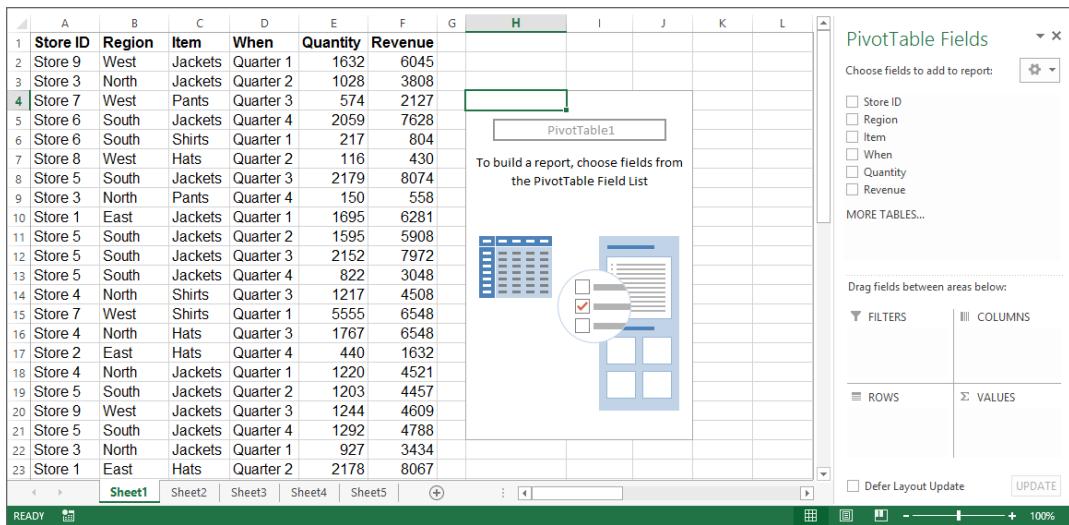


FIGURE 18-4

To create a PivotTable, complete the following steps:

1. Drag the Item field name from the Choose Fields to Add to Report pane down to the Filters pane.
2. Drag the Region field name from the Choose Fields to Add to Report pane down to the Rows pane.
3. Drag the Store ID field name from the Choose Fields to Add to Report pane down to the Rows pane, below the Region field name.
4. Drag the When field name from the Choose Fields to Add to Report pane down to the Columns pane.
5. Drag the Revenue field name from the Choose Fields to Add to Report pane down to the Values pane.

Your worksheet should look similar to Figure 18-5, with a PivotTable that shows the summary of Revenue by Quarter for each Region, with each Region showing the detail of its stores' activities. The source list could have been more than a million rows deep, and the process would still have taken Excel only a couple of moments to produce the PivotTable report.

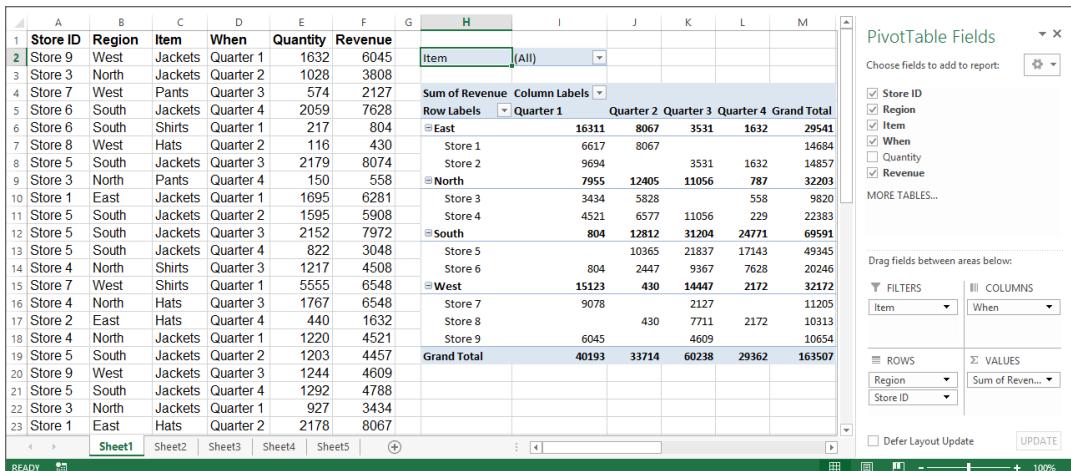


FIGURE 18-5

Hiding the PivotTable Field List

For now, you are done with the PivotTable Field List, so to clear it off your screen, you can click the X close button on its title bar, click its Ribbon icon on the PivotTable Tools Option tab, or you can right-click anywhere on the PivotTable area and select Hide Field List, as shown in Figure 18-6. When you want to see the Field List again, click the Field List Ribbon icon, or right-click anywhere on the PivotTable again and select Show Field List.

Above the PivotTable's report area, you see a small filter-looking icon in cell I2 (see Figure 18-8), in what is called the Filters area. The Item field name was dragged to that area in Step 1 of the process that created this PivotTable. If you click the filter icon, you'll see a unique list of clothing items, of which you can select one or several to have the PivotTable show only the data relating to the item(s) you select. In Figure 18-7, I selected the Hats item, and in Figure 18-8, you can see how the PivotTable adjusts itself to show only the columns and rows where data is present for the sale of hats.

The screenshot shows a Microsoft Excel spreadsheet titled "Lesson 18 - Excel". The ribbon is visible at the top with tabs like FILE, HOME, INSERT, PAGE LAYOUT, FORMULAS, DATA, REVIEW, DEVELOPER, ADD-INS, ANALYZE, and DESIGN. The ANALYZE tab is selected, showing various pivot-related tools. A PivotTable is displayed in the main area, showing data for stores across different regions and items over time. The PivotTable Fields pane is open on the right, listing fields such as Store ID, Region, Item, When, and Revenue. A context menu is open over the PivotTable Fields pane, with a red arrow pointing to the "Hide Field List" option.

FIGURE 18-6

This screenshot shows a search dialog box overlaid on a PivotTable. The dialog has a search input field with the placeholder "Search" and a dropdown menu below it containing items: (All), Hats, Jackets, Pants, Scarves, and Shirts. The "Hats" item is highlighted with a red arrow. At the bottom of the dialog are "OK" and "Cancel" buttons, with a red arrow pointing to the "OK" button. The background shows a PivotTable with columns for Quantity and Revenue, and rows for various store categories and their sub-items.

FIGURE 18-7

	A	B	C	D	E	F	G	H	I	J	K	L	M	
1	Store ID	Region	Item	When	Quantity	Revenue								
2	Store 9	West	Jackets	Quarter 1	1632	6045			Item	Hats				
3	Store 3	North	Jackets	Quarter 2	1028	3808								
4	Store 7	West	Pants	Quarter 3	574	2127			Sum of Revenue	Column Labels				
5	Store 6	South	Jackets	Quarter 4	2059	7628			Row Labels	Quarter 2	Quarter 3	Quarter 4	Grand Total	
6	Store 6	South	Shirts	Quarter 1	217	804			East		8067	3531	1632	13230
7	Store 8	West	Hats	Quarter 2	116	430			Store 1		8067			8067
8	Store 5	South	Jackets	Quarter 3	2179	8074			Store 2			3531	1632	5163
9	Store 3	North	Pants	Quarter 4	150	558			North		2020	6548		8568
10	Store 1	East	Jackets	Quarter 1	1695	6281			Store 3		2020			
11	Store 5	South	Jackets	Quarter 2	1595	5908			Store 4			6548		6548
12	Store 5	South	Jackets	Quarter 3	2152	7972			South		5791			5791
13	Store 5	South	Jackets	Quarter 4	822	3048			Store 5		5791			
14	Store 4	North	Shirts	Quarter 3	1217	4508			West		430			430
15	Store 7	West	Shirts	Quarter 1	5555	6548			Store 8		430			430
16	Store 4	North	Hats	Quarter 3	1767	6548			Grand Total		10517	15870	1632	28019
17	Store 2	East	Hats	Quarter 4	440	1632								
18	Store 4	North	Jackets	Quarter 1	1220	4521								
19	Store 5	South	Jackets	Quarter 2	1203	4457								

FIGURE 18-8

Formatting Numbers in the Values Area

You can see that the numbers in the PivotTable's Values area are unformatted. As an example of formatting them as Currency, right-click any cell in the Values area and select Value Field Settings, as indicated in Figure 18-9. In the Value Field Settings dialog box, click the Number Format button, as shown in Figure 18-10.

	E	F	G	H	I	J	K	L	M
Quantity	Revenue								
1632	6045			Item	Hats				
1028	3808								
574	2127			Sum of Revenue	Column Labels				
2059	7628			Row Labels	Quarter 2	Quarter 3	Quarter 4	Grand Total	
217	804			East		8067	3531	1632	13230
116	430			Store 1		8067			
2179	8074			Store 2			3531	1632	5163
150	558			North		2020	6548		8568
1695	6281			Store 3		2020			
1595	5908			Store 4			6548		
2152	7972			South					
822	3048			Store 5					
1217	4508			West		430			
5555	6548			Store 8		430			
1767	6548			Grand Total		10517			
440	1632								
1220	4521								
1203	4457								
1244	4609								
1292	4788								
927	3434								
2178	8067								
1563	5791								
2512	9307								
887	336								
9987	2020								
2081	7711								
61	229								
2617	9694								

FIGURE 18-9

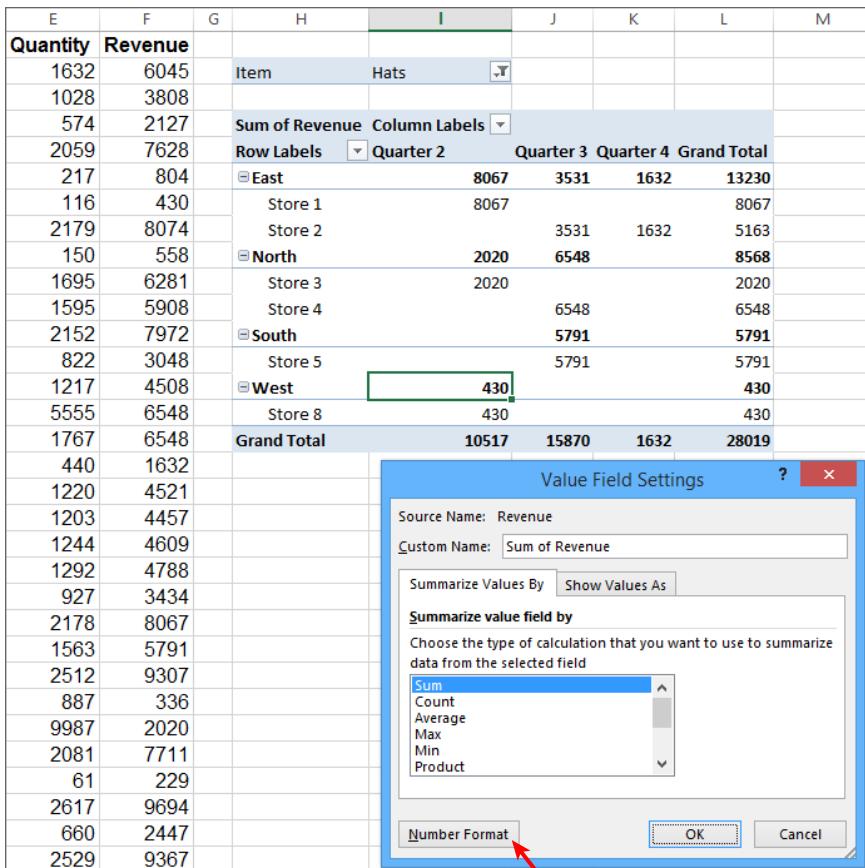


FIGURE 18-10

The familiar Format Cells dialog box appears. In Figure 18-11, I selected Currency with the dollar sign symbol and no decimal places. After you click OK in the Format Cells dialog box, you then need to click OK in the Value Field Settings dialog box, as shown in Figure 18-12.

The cells in the Values area are now formatted as Currency. Recall that earlier, the Item named Hats was selected in the Filters area. Go ahead and click the filter icon in cell I2, select the All item, and click OK, as indicated in Figure 18-13. The PivotTable report is now fully displayed with all the Values area cells formatted as Currency, including the cells that had been hidden while the Hats item was filtered.

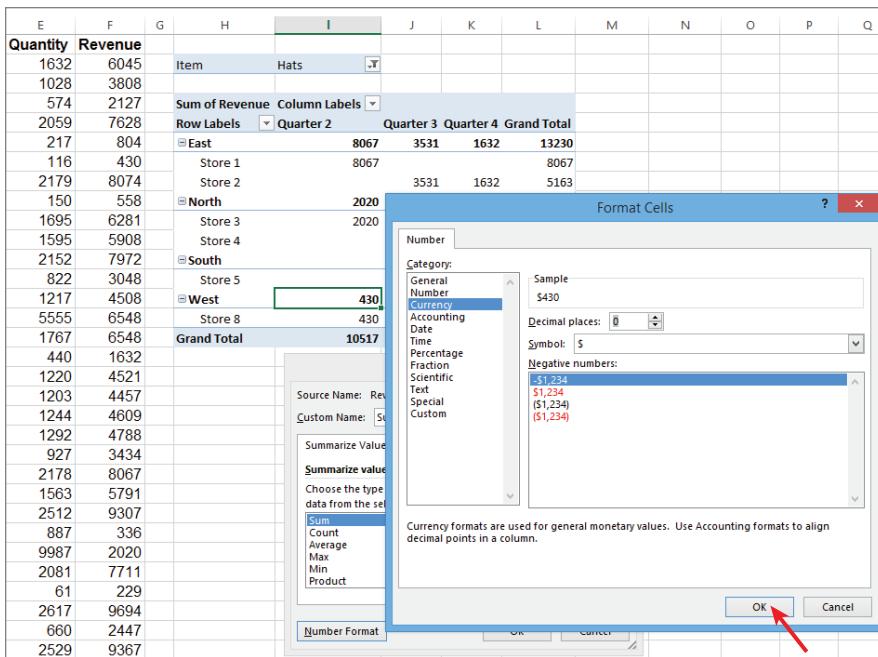


FIGURE 18-11

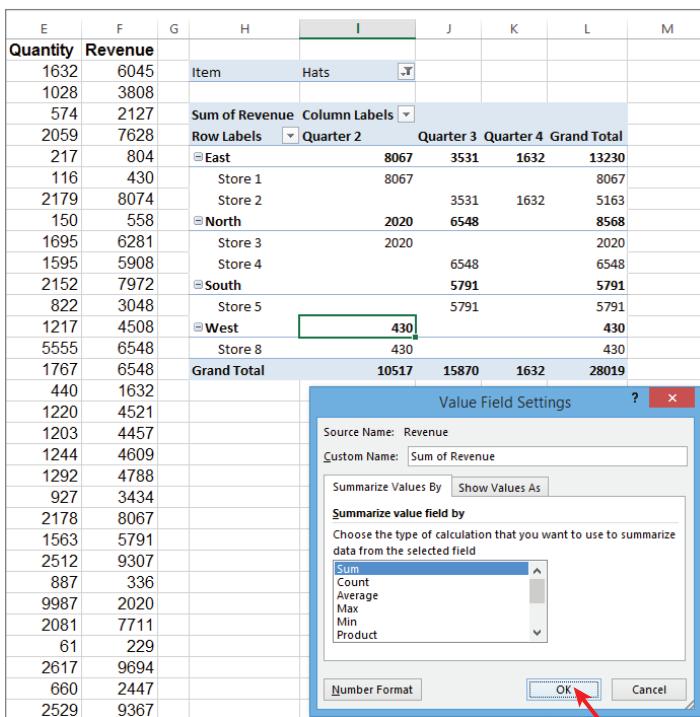


FIGURE 18-12

Quantity	Revenue	Item	Hats
1632	6045		
1028	3808		
574	2127		
2059	7628		
217	804		
116	430		
2179	8074		
150	558		
1695	6281		
1595	5908		
2152	7972		
822	3048		
1217	4508		
5555	6548		
1767	6548		
440	1632		
1220	4521		
1203	4457		
1244	4609		
1292	4788		

FIGURE 18-13

Pivoting Your Data

One of the most attractive features of a PivotTable is its ability to display the same data in whatever row-and-column arrangement of your field names that you prefer. Just as the essence of a pivot is to allow for the rotation or maneuver from a central point, you can rearrange your source data by varying the location of your field names in the row and column areas of your PivotTable.

For example, because you have summarized the clothing stores by Revenue for each Region by Quarter, you now want to look at the Quantity of each Item that was sold by Region. Reopen the PivotTable Field List and pivot your data by dragging the Item field name out of the Filters pane and into the Row Labels pane. Relocate the Region field into the Columns pane. Finally, in the Choose Fields To Add To Report pane, deselect Revenue and select Quantity. Your new PivotTable report looks like Figure 18-14.

A	B	C	D	E	F	G	H	I	J	K	L	M	N
1	Store ID	Region	Item	When	Quantity	Revenue							
2	Store 9	West	Jackets	Quarter 1	1632	6045							
3	Store 3	North	Jackets	Quarter 2	1028	3808							
4	Store 7	West	Pants	Quarter 3	574	2127	Sum of Quantity	Column Labels					
5	Store 6	South	Jackets	Quarter 4	2059	7628		Row Labels					
6	Store 6	South	Shirts	Quarter 1	217	804	Hats	North	South	West	Grand Total		
7	Store 8	West	Hats	Quarter 2	116	430	Jackets	3571	11754	1563	116	17004	
8	Store 5	South	Jackets	Quarter 3	2179	8074	Pants	2582	3175	17003	4145	26905	
9	Store 3	North	Pants	Quarter 4	150	558	Scarves	1925	574		2499		
10	Store 1	East	Jackets	Quarter 1	1695	6281	Shirts	2617			2617		
11	Store 5	South	Jackets	Quarter 2	1595	5908		1278	217	7636	9131		
12	Store 5	South	Jackets	Quarter 3	2152	7972	Grand Total	8770	18132	18783	12471	58156	
13	Store 5	South	Jackets	Quarter 4	822	3048							
14	Store 4	North	Shirts	Quarter 3	1217	4508							
15	Store 7	West	Shirts	Quarter 1	5555	6548							
16	Store 4	North	Hats	Quarter 3	1767	6548							
17	Store 2	East	Hats	Quarter 4	440	1632							
18	Store 4	North	Jackets	Quarter 1	1220	4521							
19	Store 5	South	Jackets	Quarter 2	1203	4457							

FIGURE 18-14

Creating a PivotChart

Creating a PivotChart is very easy, using either of two methods. With one method you create the chart right from the start, when you first indicate to Excel that you want to create a new PivotTable. With the other method you create a PivotChart after you have already created a PivotTable.

In Excel version 2010—shown in Figure 18-15—you can click the arrow on the lower half of the PivotTable icon on the Ribbon’s Insert tab, where an option is there for you to select PivotChart. If you want a PivotChart with your new PivotTable, just select the PivotChart option, and a PivotChart is created as you build your PivotTable in the PivotTable Field List.

To create a PivotChart as you create a new PivotTable in version 2013, from the Insert tab on the Ribbon, click the down arrow on the PivotChart icon in the Charts section. Select PivotChart & PivotTable, as shown in Figure 18-16.

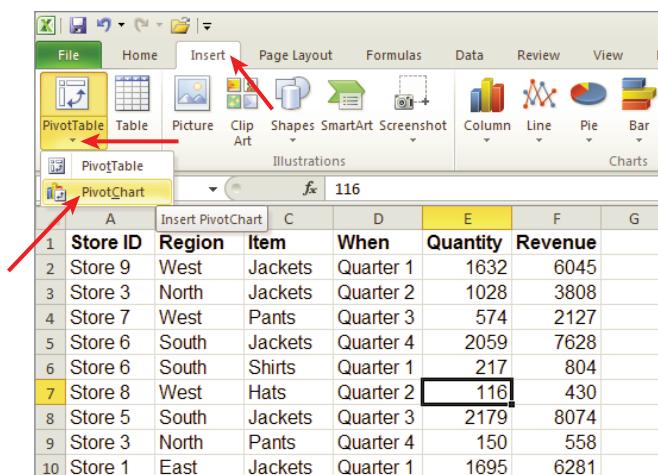


FIGURE 18-15

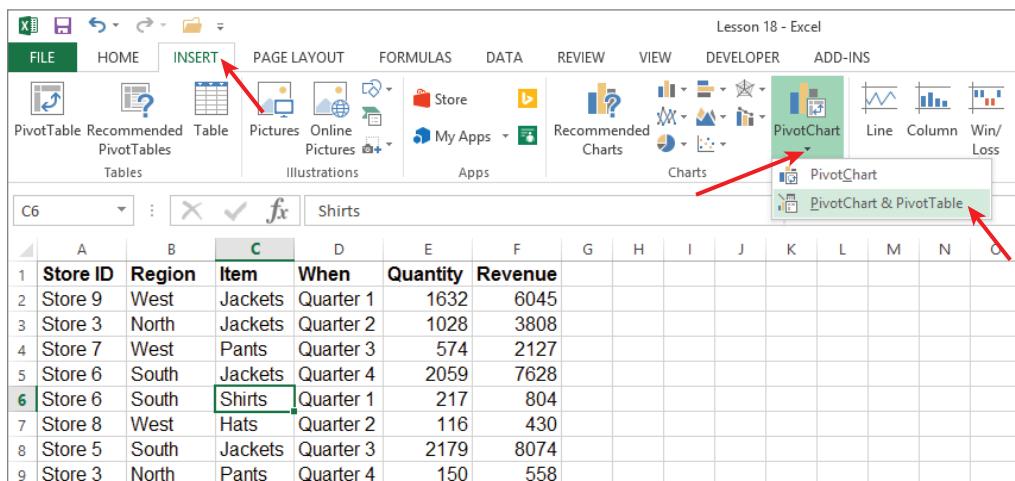


FIGURE 18-16

If you create a PivotTable and later decide you'd like a PivotChart to go along with it, you can start by selecting any cell in the PivotTable. In Excel version 2010, click the Options tab in the PivotTable Tools section of the Ribbon, and click the PivotChart icon, as shown in Figure 18-17. In version 2013, click the Analyze tab in the PivotTable Tools section of the Ribbon, and click the PivotChart icon, as shown in Figure 18-18.

Revenue							
6045							
3808							
2127							
7628	Sum of Quantity	Column Labels					
	Row Labels	East	North	South	West	Grand Total	
804	Hats		3131	2312	1563	116	7122
430	Jackets		2225	3175	17003	4145	26548
8074	Pants			1925		574	2499
558	Scarves		2617				2617
6281	Shirts		1278	217	4088		5583
5908	Grand Total		7973	8690	18783	8923	44369
7972							
3048							

FIGURE 18-17

Store ID	Region	Item	When	Quantity	Revenue
Store 9	West	Jackets	Quarter 1	1632	6045
Store 3	North	Jackets	Quarter 2	1028	3808
Store 7	West	Pants	Quarter 3	574	2127
Store 6	South	Jackets	Quarter 4	2059	7628
Store 6	South	Shirts	Quarter 1	217	804
Store 8	West	Hats	Quarter 2	116	430
Store 5	South	Jackets	Quarter 3	2179	8074
Store 3	North	Pants	Quarter 4	150	558
Store 1	East	Jackets	Quarter 1	1695	6281
Store 5	South	Jackets	Quarter 2	1595	5908
Store 5	South	Jackets	Quarter 3	2152	7972
Store 5	South	Jackets	Quarter 4	822	3048
Store 4	North	Shirts	Quarter 3	1217	4508
Store 7	West	Shirts	Quarter 1	5555	6548
Store 4	North	Hats	Quarter 3	1767	6548
Store 2	East	Hats	Quarter 4	440	1632
Store 4	North	Jackets	Quarter 1	1220	4521
Store 5	South	Jackets	Quarter 2	1203	4457

FIGURE 18-18

The Insert Chart dialog box opens, and you select your preferred chart type. Figure 18-19 shows the result after I selected the Clustered Column chart type and clicked OK. The result is a PivotChart tied to the PivotTable as shown in Figure 18-20.

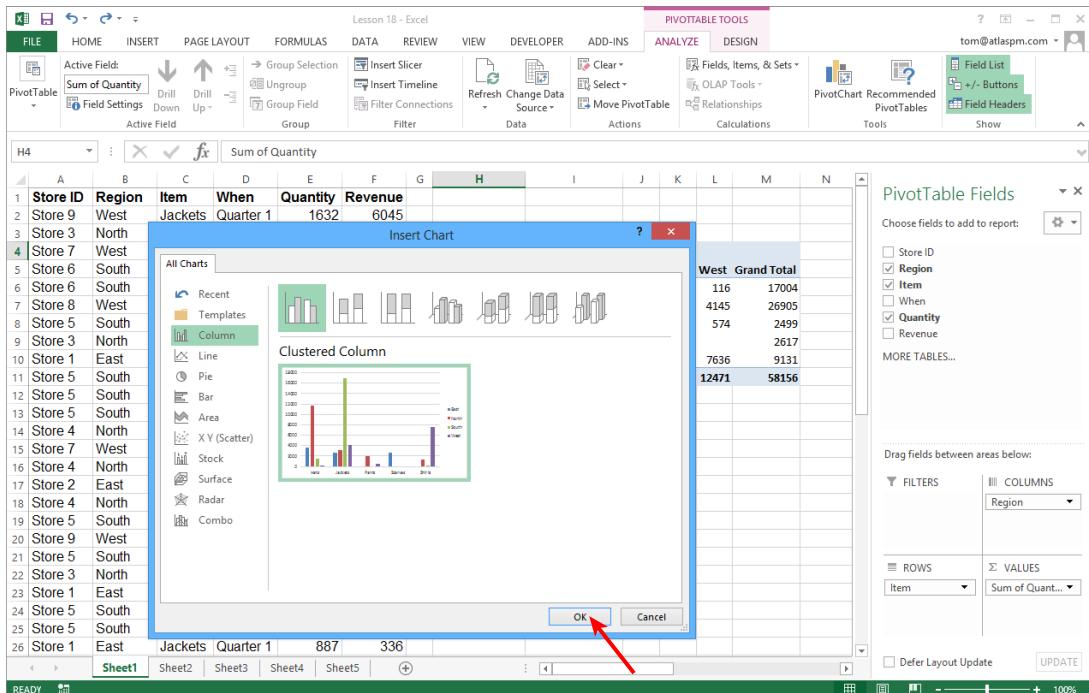


FIGURE 18-19

As you can see, when it comes to PivotCharts, Excel does almost all the grunt work for you. All you need to do is tell Excel that you want a PivotChart and what type of chart you want, and your chart is produced with its accompanying PivotTable.

NOTE There is a lot more you can do with PivotCharts and PivotTables; like many other topics, it's one that can fill an entire book. My objective so far in the lesson is to cover the basics of creating and working with PivotTables as a foundation for the VBA examples in the next sections.

PivotCharts are great—they are equipped with Field buttons so you can choose which items in which fields you want to see. Whatever field setting you select on a PivotChart makes the same change to its PivotTable. The following macro toggles between showing and hiding the Field buttons on your PivotChart:

```
Sub ShowHidePivotChartFieldButtons()
    ActiveSheet.ChartObjects(1).Activate
```

```

With ActiveChart
    .HasPivotFields = Not .HasPivotFields
End With
Range("A1").Select
End Sub

```

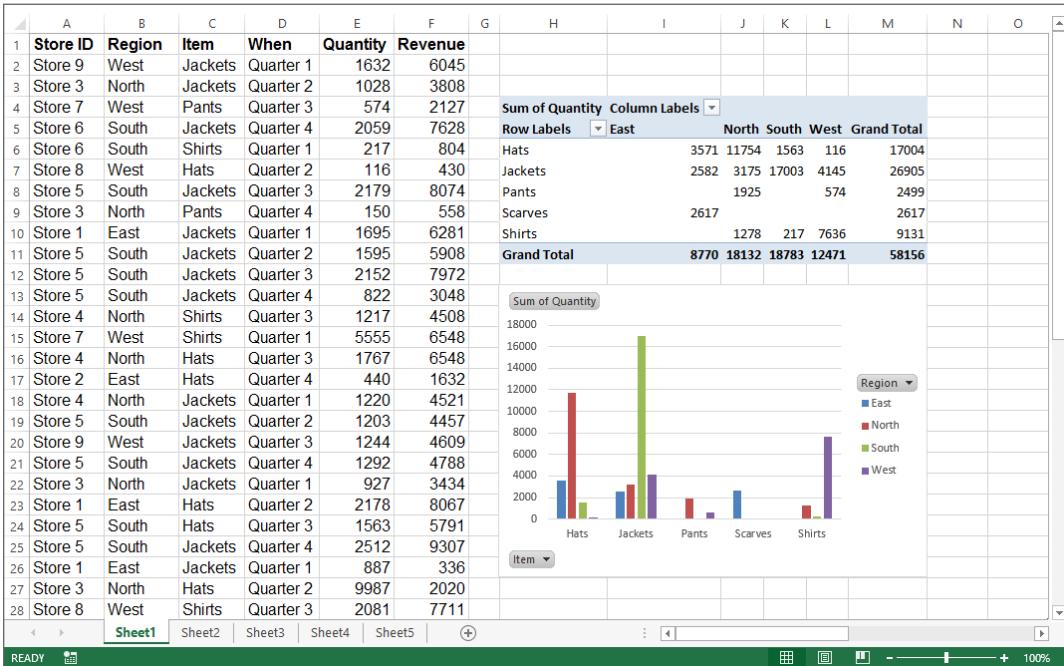


FIGURE 18-20

UNDERSTANDING PIVOTCACHES

A *PivotCache* is an object that you do not see, because it is working behind the scenes when a new PivotTable is created directly from the source data. The PivotCache is a container that holds a static copy of the source data in memory.

PivotTables do not summarize data directly from the source data, but rather from the PivotCache that memorized a snapshot of the data. That is why, in the native Excel environment not enhanced with VBA, if you change a piece of existing data in the source data range, the PivotTable report does not reflect that change until you refresh the PivotTable.

Figure 18-21 shows the Refresh menu item when you right-click a cell that is part of a PivotTable. The Refresh button actually refreshes the PivotCache.

The PivotCache, though not seen, maintains the source data beforehand in a static go-to container. Keeping the data in PivotCache memory makes pivoting and recalculations a snap, but the downside is extra workbook size and less memory for other tasks.

Quantity	Revenue										L	M	N	O	P	Q
1632	6045															
1028	3808															
574	2127															
2059	7628															
217	804	Sum of Quantity	Column Labels	Row Labels	East	North	South	West								
						3571	11754	1563								
116	430	Hats				2582	3175	17003	42							
2179	8074	Jackets														
150	558	Pants				1925										
1695	6281	Scarves				2617										
1595	5908	Shirts					1278	217	70							
2152	7972	Grand Total				8770	18132	18783	124							
822	3048															
1217	4508															
5555	6548															
1767	6548															
440	1632															
1220	4521															
1203	4457															
1244	4609															
1292	4788															
927	3434															
2178	8067															
1563	5791															

FIGURE 18-21

When you create a PivotTable manually, Excel does not bother you with the PivotCache details. If you were to create a PivotTable in VBA, you'd need to address the PivotCache issue in code. Suppose you are creating a new PivotTable based on the original source data that has been shown in this lesson. Your first step would be to program VBA to tell Excel four pieces of information:

1. You want to add a PivotCache to the workbook.
2. The location of the source data.
3. Based on items 1 and 2, create the PivotTable.
4. Specify where the PivotTable will be placed.

Assuming that the worksheet holding the source data is the active sheet, and that you want the PivotTable to be located next to the source data, the following macro would handle all those instructions:

```
Sub CreatePivot()
ThisWorkbook.PivotCaches.Add _
(SourceType:=xlDatabase, _
SourceData:=Range("A1").CurrentRegion).CreatePivotTable _
(TableDestination:="R4C" & Range("A1").CurrentRegion.Columns.Count + 2
End Sub
```

NOTE The notation "R4C" & Range("A1").CurrentRegion.Columns.Count + 2 is translated as the worksheet cell that is on row 4 of the column that is two columns to the right of the last column in the source range. Recall from earlier in the lesson that I recommend placing the top-left corner of the PivotTable on row 4, and with an empty column separating the source data and the new PivotTable.

The result you get is a PivotTable, but you'd never know by its appearance at the moment—a curious range of four cells look as if they were formatted for thin borders. In this example, the four cells are in range H4:I5, as shown in Figure 18-22.

	A	B	C	D	E	F	G	H	I	J
1	Store ID	Region	Item	When	Quantity	Revenue				
2	Store 9	West	Jackets	Quarter 1	1632	6045				
3	Store 3	North	Jackets	Quarter 2	1028	3808				
4	Store 7	West	Pants	Quarter 3	574	2127				
5	Store 6	South	Jackets	Quarter 4	2059	7628				
6	Store 6	South	Shirts	Quarter 1	217	804				
7	Store 8	West	Hats	Quarter 2	116	430				
8	Store 5	South	Jackets	Quarter 3	2179	8074				
9	Store 3	North	Pants	Quarter 4	150	558				

FIGURE 18-22

The macro is just getting started, but I wanted to show you in slow motion what is taking place under the radar when a new PivotTable is created. Actually, with the preceding macro executed, you could select one of those four cells and the PivotTable Field List would appear, inviting you to drag fields to your desired location, as shown in Figure 18-23. In Figure 18-24, the Item field was moved to the Rows area and the When field was moved to the Columns area.

The screenshot shows a Microsoft Excel spreadsheet with a PivotTable Fields ribbon tab selected. The main table contains data for stores, regions, items, and quarters. A PivotTable is being created on the right side of the screen. The PivotTable Fields pane is open, showing fields for Store ID, Region, Item, When, Quantity, and Revenue. The 'When' field is currently selected and highlighted with a green border. The pane also includes sections for Filters, Columns, Rows, and Values, along with a 'Defer Layout Update' button and an 'UPDATE' button.

	A	B	C	D	E	F	G	H	I	J	K	L
1	Store ID	Region	Item	When	Quantity	Revenue						
2	Store 9	West	Jackets	Quarter 1	1632	6045						
3	Store 3	North	Jackets	Quarter 2	1028	3808						
4	Store 7	West	Pants	Quarter 3	574	2127						
5	Store 6	South	Jackets	Quarter 4	2059	7628						
6	Store 6	South	Shirts	Quarter 1	217	804						
7	Store 8	West	Hats	Quarter 2	116	430						
8	Store 5	South	Jackets	Quarter 3	2179	8074						
9	Store 3	North	Pants	Quarter 4	150	558						
10	Store 1	East	Jackets	Quarter 1	1695	6281						
11	Store 5	South	Jackets	Quarter 2	1595	5908						
12	Store 5	South	Jackets	Quarter 3	2152	7972						
13	Store 5	South	Jackets	Quarter 4	822	3048						
14	Store 4	North	Shirts	Quarter 3	1217	4508						
15	Store 7	West	Shirts	Quarter 1	5555	6548						
16	Store 4	North	Hats	Quarter 3	1767	6548						
17	Store 2	East	Hats	Quarter 4	440	1632						
18	Store 4	North	Jackets	Quarter 1	1220	4521						
19	Store 5	South	Jackets	Quarter 2	1203	4457						
20	Store 9	West	Jackets	Quarter 3	1244	4609						

FIGURE 18-23

When a numerical field is moved into the Values area, the PivotTable becomes more recognizable. For example, Figure 18-25 shows the result of moving the Revenue field into the Values area.

NOTE If you want your PivotTable's PivotCache to refresh automatically when a cell in your source list changes, the following Worksheet_Change event handles that. Note that the code uses the PivotTable's Index property for the first or only PivotTable on the worksheet to be refreshed:

```
Private Sub Worksheet_Change(ByVal Target As Range)
If Intersect(Target, Range("A1").CurrentRegion) Is Nothing Then
    Exit Sub
End If
If Target.Cells.Count > 1 Then Exit Sub
ActiveSheet.PivotTables(1).PivotCache.Refresh
End Sub
```

Store ID	Region	Item	When	Quantity	Revenue
Store 9	West	Jackets	Quarter 1	1632	6045
Store 3	North	Jackets	Quarter 2	1028	3808
Store 7	West	Pants	Quarter 3	574	2127
Store 6	South	Jackets	Quarter 4	2059	7628
Store 6	South	Shirts	Quarter 1	217	804
Store 8	West	Hats	Quarter 2	116	430
Store 5	South	Jackets	Quarter 3	2179	8074
Store 3	North	Pants	Quarter 4	150	558
Store 1	East	Jackets	Quarter 1	1695	6281
Store 5	South	Jackets	Quarter 2	1595	5908
Store 5	South	Jackets	Quarter 3	2152	7972
Store 5	South	Jackets	Quarter 4	822	3048
Store 4	North	Shirts	Quarter 3	1217	4508
Store 7	West	Shirts	Quarter 1	5555	6548
Store 4	North	Hats	Quarter 3	1767	6548
Store 2	East	Hats	Quarter 4	440	1632
Store 4	North	Jackets	Quarter 1	1220	4521
Store 5	South	Jackets	Quarter 2	1203	4457
Store 9	West	Jackets	Quarter 3	1244	4609

FIGURE 18-24

Store ID	Region	Item	When	Quantity	Revenue	Sum of Revenue	When
Store 9	West	Jackets	Quarter 1	1632	6045		
Store 3	North	Jackets	Quarter 2	1028	3808		
Store 7	West	Pants	Quarter 3	574	2127		
Store 6	South	Jackets	Quarter 4	2059	7628		
Store 6	South	Shirts	Quarter 1	217	804		
Store 8	West	Hats	Quarter 2	116	430		
Store 5	South	Jackets	Quarter 3	2179	8074		
Store 3	North	Pants	Quarter 4	150	558		
Store 1	East	Jackets	Quarter 1	1695	6281		
Store 5	South	Jackets	Quarter 2	1595	5908		
Store 5	South	Jackets	Quarter 3	2152	7972		
Store 5	South	Jackets	Quarter 4	822	3048		
Store 4	North	Shirts	Quarter 3	1217	4508		
Store 7	West	Shirts	Quarter 1	5555	6548		
Store 4	North	Hats	Quarter 3	1767	6548		
Store 2	East	Hats	Quarter 4	440	1632		
Store 4	North	Jackets	Quarter 1	1220	4521		
Store 5	South	Jackets	Quarter 2	1203	4457		
Store 9	West	Jackets	Quarter 3	1244	4609		

FIGURE 18-25

MANIPULATING PIVOTFIELDS IN VBA

PivotFields are the row and column areas that you place your field names into, depending on how you want the PivotTable to display your data. The following pieces of VBA code perform the placement of PivotFields as they were for the PivotTable that you created manually earlier in the lesson. Two fields (Region and Store ID) are placed as row labels, and one field (When) is placed as a column label. The Revenue field is placed in the Values area, and the Filters area is populated by the Items field:

```
With ActiveSheet.PivotTables(1)

    'First (outer) row field.
    With .PivotFields("Region")
        .Orientation = xlRowField
        .Position = 1
    End With

    'Second (inner) row field.
    With .PivotFields("Store ID")
        .Orientation = xlRowField
        .Position = 2
    End With

    'Column field.
    With .PivotFields("When")
        .Orientation = xlColumnField
        .Position = 1
    End With

    'Filters area.
    With .PivotFields("Item")
        .Orientation = xlPageField
        .Position = 1
    End With

    'Revenue in the Values field.
    .AddDataField ActiveSheet.PivotTables(1).PivotFields("Revenue"), _
    "Sum of Amount", xlSum

End With
```

NOTE Be sure to name your PivotFields correctly! They must be spelled the same way in your code as they are in the header cells of your source list. If you misspell the field names in your code, VBA lets you know with a runtime error because the field names you're instructing VBA to manipulate do not exist.

MANIPULATING PIVOTITEMS WITH VBA

PivotItems are programmable in PivotTables, and as an example, you can arrange to see just one particular PivotItem in a field. In a PivotTable that you created earlier in the lesson, you added a Region field. Suppose you want to see activity only for the North PivotItem and hide the South, East, and West PivotItems. The following macro accomplishes that:

```
Sub ShowSingleItem()
    Dim objPivotField As PivotField
    Dim objPivotItem As PivotItem
    Set objPivotField = _
        ActiveSheet.PivotTables(1).PivotFields(Index:="Region")
    For Each objPivotItem In objPivotField.PivotItems
        If objPivotItem.Name = "North" Then
            objPivotItem.Visible = True
        Else
            objPivotItem.Visible = False
        End If
    Next objPivotItem
End Sub
```

The following macro shows all the PivotItems:

```
Sub ShowAllItems()
    Dim objPivotField As PivotField
    Dim objPivotItem As PivotItem
    Set objPivotField = _
        ActiveSheet.PivotTables(1).PivotFields(Index:="Region")
    For Each objPivotItem In objPivotField.PivotItems
        objPivotItem.Visible = True
    Next objPivotItem
End Sub
```

CREATING A PIVOTTABLES COLLECTION

PivotTables are objects for which there is a `Collection` object, just as there is for worksheets and workbooks. As you might guess, the name of the `Collection` object for PivotTables is `PivotTables`, and you can loop through every PivotTable on a worksheet or throughout the workbook if you need to.

For example, if you have more than one PivotTable on a worksheet and they are tied to the same source list that starts in cell A1, this `Worksheet_Change` event refreshes all PivotTables on that worksheet automatically when the source data is changed:

```
Private Sub Worksheet_Change(ByVal Target As Range)
    If Intersect(Target, Range("A1").CurrentRegion) Is Nothing _
        Or Target.Cells.Count > 1 Then Exit Sub
```

```
Dim PT As PivotTable
For Each PT In ActiveSheet.PivotTables
    PT.RefreshTable
Next PT
End Sub
```

Suppose you have several PivotTables on many different worksheets and you want to be confident that every PivotTable displays the current data from its respective source lists. The following Workbook_Open procedure refreshes every PivotTable in the workbook when the workbook opens:

```
Private Sub Workbook_Open()
    Dim wks As Worksheet, PT As PivotTable
    For Each wks In Worksheets
        For Each PT In wks.PivotTables
            PT.RefreshTable
        Next PT
    Next wks
End Sub
```

NOTE You can avoid looping through all your PivotTables by using VBA's RefreshAll method to refresh all PivotTables at once. The single line of code would be ActiveWorkbook.RefreshAll. Just be aware that the RefreshAll method also refreshes all external data ranges, such as web queries, for the specified workbook.

You might need to delete all the PivotTables on a worksheet. When you delete a PivotTable, what you are really doing is clearing the cells that are occupied by the PivotTable. The following macro deletes all the PivotTables on the active worksheet:

```
Sub DeleteAllPivotTablest()
    Dim objPT As PivotTable, iCount As Integer
    For iCount = ActiveSheet.PivotTables.Count To 1 Step -1
        Set objPT = ActiveSheet.PivotTables(iCount)
        objPT.PivotSelect ""
        Selection.Clear
    Next iCount
End Sub
```

TRY IT

In this lesson, you write a macro that adds a PivotChart to accompany an existing PivotTable. Your new PivotChart will be located below the PivotTable on that same worksheet.

Lesson Requirements

Your worksheet contains a list of source data, and you already have a PivotTable on your worksheet, as shown in Figure 18-26. To get the sample workbook, you can download Lesson 18 from the book's website at www.wrox.com/go/excelvba24hour.

	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
1	Store ID	Region	Item	When	Quantity	Revenue									
2	Store 9	West	Jackets	Quarter 1	1632	6045									
3	Store 3	North	Jackets	Quarter 2	1028	3808									
4	Store 7	West	Pants	Quarter 3	574	2127									
5	Store 6	South	Jackets	Quarter 4	2059	7628									
6	Store 6	South	Shirts	Quarter 1	217	804									
7	Store 8	West	Hats	Quarter 2	116	430									
8	Store 5	South	Jackets	Quarter 3	2179	8074									
9	Store 3	North	Pants	Quarter 4	150	558									
10	Store 1	East	Jackets	Quarter 1	1695	6281									
11	Store 5	South	Jackets	Quarter 2	1595	5908									
12	Store 5	South	Jackets	Quarter 3	2152	7972									
13	Store 5	South	Jackets	Quarter 4	822	3048									
14	Store 4	North	Shirts	Quarter 3	1217	4508									
15	Store 7	West	Shirts	Quarter 1	5555	6548									
16	Store 4	North	Hats	Quarter 3	1767	6548									
17	Store 2	East	Hats	Quarter 4	440	1632									
18	Store 4	North	Jackets	Quarter 1	1220	4521									
19	Store 5	South	Jackets	Quarter 2	1203	4457									
20	Store 9	West	Jackets	Quarter 3	1244	4609									
21	Store 5	South	Jackets	Quarter 4	1292	4788									
22	Store 3	North	Jackets	Quarter 1	927	3434									
23	Store 1	East	Hats	Quarter 2	2178	8067									
24	Store 5	South	Hats	Quarter 3	1563	5791									
25	Store 5	South	Jackets	Quarter 4	2512	9307									
26	Store 1	East	Jackets	Quarter 1	887	336									
27	Store 3	North	Hats	Quarter 2	9987	2020									
28	Store 8	West	Shirts	Quarter 3	2081	7711									
29	Store 4	North	Shirts	Quarter 4	61	229									
30	Store 2	East	Scarves	Quarter 1	2617	9694									
31	Store 6	South	Jackets	Quarter 2	660	2447									
32	Store 6	South	Jackets	Quarter 3	2529	9367									
33	Store 8	West	Jackets	Quarter 4	586	2172									
34	Store 7	West	Jackets	Quarter 1	683	2530									
35	Store 4	North	Pants	Quarter 2	1775	6577									
36	Store 2	East	Hats	Quarter 3	953	3531									

FIGURE 18-26

Step-by-Step

1. Activate the worksheet that contains the source data list and PivotTable.
2. Press Alt+F11 to go to the Visual Basic Editor.
3. From the menu bar, click Insert ➔ Module.
4. In the new module, type `Sub CreatePivotChart` and press Enter. VBA produces the following lines of code for you:

```
Sub CreatePivotChart()

End Sub
```

5. Turn off ScreenUpdating to help your macro run faster by not refreshing the screen as objects in the code are created and manipulated:

```
Application.ScreenUpdating = False
```

6. Declare an Object variable for the existing PivotTable:

```
Dim objPT As PivotTable
```

7. Set the Object variable for the first (index #1) PivotTable:

```
Set objPT = ActiveSheet.PivotTables(1)
```

8. Select the PivotTable:

```
objPT.PivotSelect ""
```

9. Add the chart:

```
Charts.Add
```

10. Place the chart onto the PivotTable's worksheet:

```
ActiveChart.Location Where:=xlLocationAsObject, _  
Name:=objPT.Parent.Name
```

11. Position the PivotChart so its top-left corner occupies cell H23, a few rows below the PivotTable:

```
ActiveChart.Parent.Left = Range("H23").Left  
ActiveChart.Parent.Top = Range("H23").Top
```

12. Deselect the PivotChart:

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

13. Turn on ScreenUpdating:

```
Application.ScreenUpdating = True
```

14. When you complete the macro, it looks as follows:

```
Sub CreatePivotChart()  
  
    'Turn off ScreenUpdating.  
    Application.ScreenUpdating = False  
  
    'Declare an Object variable for the existing PivotTable.  
    Dim objPT As PivotTable  
    'Set the Object variable for the first (index #1) PivotTable.  
    Set objPT = ActiveSheet.PivotTables(1)  
  
    'Select the PivotTable.  
    objPT.PivotSelect ""  
  
    'Add the chart.  
    Charts.Add  
  
    'Place it on the PivotTable's worksheet.  
    ActiveChart.Location Where:=xlLocationAsObject, _  
    Name:=objPT.Parent.Name  
  
    'Position the PivotChart so its top left corner  
    'occupies cell H23, a few rows below the PivotTable.  
    ActiveChart.Parent.Left = Range("H23").Left
```

```

ActiveChart.Parent.Top = Range("H23").Top

'Deselect the PivotChart.
Range("A1").Select

'Turn on ScreenUpdating.
Application.ScreenUpdating = True

End Sub

```

- 15.** Press Alt+Q to return to your worksheet and test your macro. Figure 18-27 shows what the worksheet should look like with the PivotChart added, right where it was specified in VBA.

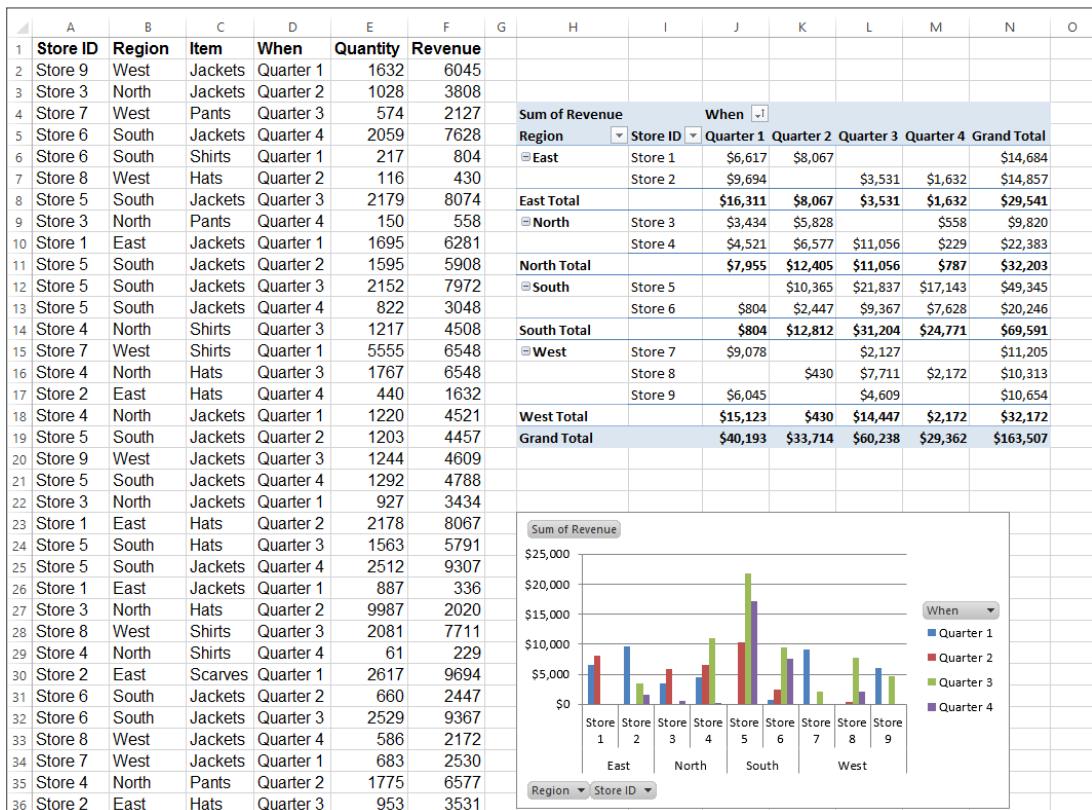


FIGURE 18-27

REFERENCE Please select the video for Lesson 18 at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

19

User-Defined Functions

Most Excel users who are not absolute beginners use worksheet functions in their formulas. The most common worksheet function is the `SUM` function, and hundreds more exist.

Basically, a function performs a calculation or evaluation and then returns a value. Functions used in your VBA expressions act the same way; they do what they are programmed to do and then return a result.

With VBA, you can write (“define”) your own custom function (a user-defined function or UDF) that looks, acts, and feels like a built-in function, but with a lot more power and versatility. After you get the hang of UDFs, you’ll wonder how you ever got along without them.

WHAT IS A USER-DEFINED FUNCTION?

You are already familiar with many of Excel’s built-in worksheet functions such as `SUM`, `AVERAGE`, and `VLOOKUP`, but sometimes you need to perform calculations or get information that none of Excel’s built-in functions can accomplish. A *user-defined function (UDF)* is a function in VBA that you create with arguments you specify. You use it as a worksheet function or as part of a macro procedure when a task is otherwise impossible or too cumbersome to achieve with Excel’s built-in formulas and functions.

For example, you may need a formula to sum a range of numbers depending on a cell’s interior color; to extract only numbers or letters from an alphanumeric string; to place an unchanging random number in a cell; or to test whether a particular worksheet exists or another workbook is open. UDFs are an excellent option for handling tasks when regular worksheet functions cannot or should not be used.

Characteristics of User-Defined Functions

When used as a worksheet function, the purpose of a UDF is to return a number, string, array, or boolean (true or false) value to the cell it occupies. UDFs cannot change the Excel environment in any way, meaning they cannot place a value in another cell, change the interior color

of any cell including the cell they are in, rename a worksheet, or do anything other than return a value to their own cell.

That said, it's important to note that a UDF can be called by a macro. This allows the calling procedure (the macro) to take advantage of the UDF while still retaining the ability to change the Excel environment. This makes your UDF a versatile tool when integrated with macros.

UDFs cannot be composed by the Macro Recorder. Although in some cases you can record a macro and turn it into a UDF by editing the code, most of the time you create a UDF by writing the code yourself directly into a standard module.

NOTE *UDFs are always located in a standard module, though they can neither appear in, nor be run from, the Macro dialog box. UDFs do not work if placed in any other type of module, such as a worksheet, workbook, UserForm, or class module.*

NOTE *Whichever way the UDF is called, be aware that it always compiles slower than built-in functions. Honestly, you would barely notice the difference yourself, but Excel notices and spends an extra moment to think things over when compiling code of any kind, including user-defined functions. Avoid reinventing the wheel by using worksheet functions wherever practical and using UDFs only for things worksheet functions cannot or should not do.*

Anatomy of a UDF

When designing a UDF, it helps to consider three questions:

- What is the function's purpose; that is, what do you want it to accomplish?
- What arguments, if any, does the function need?
- What will the function return as a formula or provide to its caller in a macro?

A UDF always begins with the `Function` statement and ends with the `End Function` statement. Unless you want your function to be visible only to other code in the same module, it's best to declare the function as `Public`, or omit the `Public/Private` qualifier altogether, which defaults the function's scope to `Public`. When you declare a UDF as `Public`, it appears in the list of functions in the Insert Function dialog box.

Here is an example of the general syntax of a UDF:

```
Function name([argument list]) as type
'VBA statements that make up the Function
[name = returned expression]
End Function
```

NOTE Function names must begin with a letter, and cannot contain spaces or illegal naming characters such as the slash, colon, comma, bracket, or any arithmetic operator symbols. It's always a good practice to give your UDF a simple, meaningful name, just as you would for a macro.

After the function's name is the argument list, which is enclosed by parentheses. If you have two or more arguments, each is separated by a comma and a space. Not every UDF requires arguments, but the parentheses are still required immediately after the function name. Following the argument list is the optional (but strongly recommended) specification of the data type, depending on the function's purpose.

Here's an example of a UDF that does not require any arguments. It returns the complete path of the Microsoft Excel application on your computer:

```
Function xlPath() As String
    xlPath = Application.Path
End Function
```

On my computer, using Microsoft Office 2013 and entering the formula `=xlPath()` into a worksheet cell, this UDF returns the path `C:\ProgramFiles\Microsoft Office 15\root\office15.`

UDF EXAMPLES THAT SOLVE COMMON TASKS

User-defined functions can simplify your work by enabling you to use shorter and more readable formulas. After you create the UDF, all the user needs to know is the function name and its arguments. User-defined functions are very useful for handling everyday tasks that you might have thought—or are known to be impossible to solve with a native worksheet formula. Following are a few examples of UDFs that can solve such tasks.

Summing Numbers in Colored Cells

A question that frequently arises is how to add up the numbers that are only in colored cells of a certain range. If the cells were colored by conditional formatting, the solution could be to sum that range of cells based on the condition, such as by using the `SUMIF` function. However, evaluating the property of a cell—in this case its actual interior color—is more of a challenge because no built-in worksheet function is able to do that.

As an example, Figure 19-1 shows a list of numbers in range A2:A15, where some cells are colored gray and some are not. The task is to sum the numbers in gray-colored cells.

Outside the range, cell C1 serves the dual purpose of receiving the UDF, and also displaying the color you need to sum by. With this approach, the UDF only needs one argument to specify the range to sum:

```

Function SumColor(RangeToSum As Range) As Long
'Declare the necessary variables.
Dim ColorID As Integer, ColorCell As Range, mySum As Long
'Identify the ColorID variable so you know what color to look for.
ColorID = Range(Application.Caller.Address).Interior.ColorIndex
'Loop through each cell in the range.
For Each ColorCell In RangeToSum
'If the cell's color matches the color you are looking for,
'keep a running subtotal by adding the cell's number value
'to the mySum variable.
If ColorCell.Interior.ColorIndex = ColorID Then mySum = mySum + ColorCell.Value
Next ColorCell
'The cells have all been evaluated, so you can define the SumColor function
'by setting it equal to the mySum variable.
SumColor = mySum
End Function

```

C1	A	B	C	D	E
1	Numbers		16		
2	7				
3	4				
4	3				
5	6				
6	6				
7	1				
8	9				
9	4				
10	1				
11	3				
12	7				
13	1				
14	2				
15	6				

FIGURE 19-1

The entry in cell C1 is =SumColor(A2:A15). The UDF loops through each cell in range A2:A15, and along the way keeps a running total with the mySum variable when a gray cell is encountered. At the end of the UDF code, the function's name of SumColor is set to equal the mySum variable, and that enables the UDF to return 16 as the sum of gray-colored cells. Notice that because you were expecting the result to be a whole number, the Long variable type was specified for the function's name.

NOTE This example also demonstrates another useful way to employ the Application.Caller statement that's introduced in Lesson 16. Here, the object calling the function is cell C1, which was colored gray before the UDF was entered.

Extracting Numbers or Letters from an Alphanumeric String

Another common question is how to extract numbers or letters from a string that contains a mixture of alphanumeric characters. If the numbers or letters are all in predictable places or consistently grouped in some way, built-in formulas might do the job. But it gets dicey if the string has an unpredictable mishmash of characters similar to what is in column A in Figure 19-2.

	A	B	C
1	Original string	Numbers extracted	Letters extracted
2	KH5Y3W84A	5384	KHYWA
3	JU83d7x62KBV	83762	JUDxKBV
4	3L4SWc9E179	349179	LSWcE
5			
6			
7	The UDF in cell B2 and copied down is =ExtractNumbers(A2)	The UDF in cell C2 and copied down is =ExtractLetters(A2)	
8			
9			
10			

FIGURE 19-2

Following are two similar UDFs: one that extracts just the numbers from an alphanumeric string and one that extracts just the letters. Figure 19-2 shows how the formulas should be entered.

NOTE Remember that you can copy and paste a UDF just as you can a built-in formula or function. You can also use the fill handle to copy the UDF down or across.

```

Function ExtractNumbers(strText As String)
'Declare the necessary variables.
Dim i As Integer, strDbl As String
'Loop through each character in the cell.
For i = 1 To Len(strText)
    'If the character is a digit, append it to the strDbl variable.
    If IsNumeric(Mid(strText, i, 1)) Then
        strDbl = strDbl & Mid(strText, i, 1)
    End If
Next i
'Each character in the cell has been evaluated, so you can define the
'ExtractNumbers function by setting it equal to the strDbl variable.
'The purpose of the CDbL function is to coerce the strDbl expression
'into a numeric Double data type.
ExtractNumbers = CDbl(strDbl)
End Function

Function ExtractLetters(strText As String)
'Declare the necessary variables.
Dim x As Integer, strTemp As String
'Loop through each character in the cell.
For x = 1 To Len(strText)

```

```

'If the character is not numeric, it must be a letter,
'so append it to the strTemp variable.
If Not IsNumeric(Mid(strText, x, 1)) Then
    strTemp = strTemp & Mid(strText, x, 1)
End If
Next x
'Each character in the cell has been evaluated, so you can define the
'ExtractLetters function by setting it equal to the strTemp variable.
ExtractLetters = strTemp
End Function

```

Extracting the Address from a Hyperlink

Here is an example of how to return the actual underlying address of a hyperlink. In Figure 19-3, hyperlinks are in column A but the display text in those cells describes the link's destination. This UDF returns the actual hyperlink address; the "mailto:" portion of the code deals with the possibility of a link being an e-mail address:

```

Function Link(HyperlinkCell As Range)
Link = Replace(HyperlinkCell.Hyperlinks(1).Address, "mailto:", "")
End Function

```

	A	B
1	Hyperlink in cell	Hyperlink address
2	Tom Urtis' website	http://www.atlaspm.com/
3	Excel Aptitude Test	http://xat.atlaspm.com/
4		
5		The UDF in cell B2 and copied down is =Link(A2)

FIGURE 19-3

USER-DEFINED FUNCTIONS AND ERRORS

You might wonder what happens if an error occurs with a UDF. For example, what if the `SumColor` function is entered into a cell with an illogical range argument address such as `=SUMCOLOR (A2 :WXYZ)`? Or, what if a UDF attempts to divide a number by zero?

When a UDF attempts to do what it cannot do, the cell displays a #VALUE! error. Whereas a failed macro results in a runtime error with an imposing message box to announce the error and a debug option to identify the offending code line, such is not the case with a failed UDF. Even though it is a VBA item, a failed UDF only returns the #VALUE! error. With larger UDFs, finding the cause of the error can be a real chore. Therefore, it's a good idea to test each code line in the Immediate window as you write your larger UDFs.

VOLATILE FUNCTIONS

Sometimes, you want a UDF to return a value and then do nothing else until you purposely cause it to recalculate. An example is if you want to produce a random number in a cell but keep that number constant until you decide to change it again, if ever. The worksheet function `RAND()` returns a random number, but it recalculates whenever the worksheet recalculates or any cell in that worksheet is edited. This UDF returns an unchanging (static) random number between 1 and 100:

```
Function StaticRandom() As Double
StaticRandom = Int(Rnd() * 100)
End Function
```

The function entry for the cell is `=StaticRandom()`.

NOTE Notice that the `StaticRandom` UDF does not require an argument. Even so, the empty parentheses must immediately follow the function's name in the first code line. Also, when you enter a non-argument UDF in a cell, the parentheses must be included, as you see in this example.

Now with the `StaticRand` UDF in its current state, its returned random number does not change unless you purposely call the UDF, such as if you select the cell, press the F2 key, and press Enter, or if you press `Ctrl+Alt+F9` to force a calculation on all cells.

If you prefer to have the UDF act as the built-in `RAND` function would, that is, to recalculate whenever another worksheet formula is recalculated or a cell is edited, you can insert the statement `Application.Volatile` like so:

```
Function StaticRandom() As Double
Application.Volatile
StaticRandom = Int(Rnd() * 100)
End Function
```

NOTE Be aware that if the UDF is used in a lot of cells, `Application.Volatile` adds to the workbook's overall calculation effort, possibly resulting in longer recalculation times.

Returning the Name of the Active Worksheet and Workbook

A very common request is for a formula to return the name of the active worksheet or workbook. This is a case where a UDF is still a worthy alternative even though formulas can handle this request, and the `Application.Volatile` statement would be included.

For the worksheet name, this formula is an option but it's not easy to memorize or to enter correctly:

```
=MID(CELL("filename",A1),FIND("]",CELL("filename",A1))+1,32)
```

Although the formula automatically updates itself when a sheet tab name changes, the workbook must be named (saved at least once) or the formula returns a #VALUE! error.

The following code shows a UDF with the Application.Volatile statement that covers all the bases. It updates itself when the worksheet tab changes, and the workbook does not need to be named or saved for the UDF to work. Another advantage is that the formula =SheetName() is easy to remember and to enter:

```
Function SheetName() As String
Application.Volatile
SheetName = ActiveSheet.Name
End Function
```

For the formula that returns the active workbook's name, the following is a lengthier and more difficult one to enter properly:

```
=MID(CELL("filename",A1),FIND("[",CELL("filename",A1))+1,FIND("]",CELL("filename",A1))-FIND("[",CELL("filename",A1))-1)
```

The workbook needs to be saved at least once for this formula to work.

The NameWB() function is much easier to remember and enter, and it'll also do the job whether or not the workbook has been saved:

```
=NameWB()
```

Its UDF is the following:

```
Function NameWB() As String
Application.Volatile
NameWB = ActiveWorkbook.Name
End Function
```

UDFs with Conditional Formatting

One of the less-utilized but powerful applications of a UDF is to combine it with conditional formatting. Let's say you want to identify cells that contain a comment in a workbook where the option to show comment indicators is turned off. It's true that cells containing comments fall into the category of SpecialCells and you can select them through the Go To Special dialog box; you can maybe format the selected comment-containing cells from there. However, you'd need to repeat those steps anytime a cell obtains or deletes a comment, and there's no telling if or when that might happen.

A better way to go is with a UDF as the formula rule with conditional formatting, to format the comment-containing cells in real time as comments are added or deleted. For example, place this UDF into a standard module:

```
Public Function TestComment(rng As Range) As Boolean
TestComment = Not rng.Comment Is Nothing
End Function
```

Back on your worksheet, select the range of interest—in this example starting from cell A1. In the New Formatting Rule dialog box for Excel versions starting with 2007, or the Conditional Formatting dialog box for versions prior to 2007, enter this formula:

```
=TestComment(A1)
```

Choose your formatting style, click OK, and all comment-containing cells in that range are formatted.

Calling Your UDF from a Macro

As mentioned earlier, functions that you create need not only serve as worksheet formulas. A function can also be called by a macro, which does not limit the macro's ability to do whatever needs to be done. In the following code, the OpenTest function is set apart from the OpenOrClosed macro, which gives you the best of both worlds for testing whether a particular workbook is open or closed.

To test by formula if a workbook named `YourWorkbookName.xlsm` is open or closed, you can enter the following in a worksheet cell, which returns TRUE (the workbook is open) or FALSE (the workbook is closed):

```
=OpenTest ("YourWorkbookName.xlsm")
```

To test by macro, you can expand the functionality by asking with a Yes/No message box if you'd like to open that workbook if it is not already open, and open it if Yes is selected, or keep the workbook closed if No is selected. Here's the code:

```
Function OpenTest(wb) As Boolean
'Declare a Workbook variable.
Dim wkb As Workbook
'Employ the On Error Resume Next statement to check for, and bypass,
'a run time error in case the workbook is not open.
On Error Resume Next
Set wkb = Workbooks(wb)
'If there is no error, the workbook is open.
If Err = 0 Then
    Err.Clear
    OpenTest = True
Else
    'An error was raised, meaning the workbook is not open.
    OpenTest = False
End If
End Function

Sub OpenOrClosed()

'Declare a String type variable that will be the workbook name.
Dim strFileName As String
strFileName = "YourWorkbookName.xlsm"

'Call the OpenTest UDF to evaluate whether or not the workbook is open.
If OpenTest(strFileName) = True Then

    'For demo purposes, this message box informs you if the workbook is open.
    MsgBox strFileName & " is open.", vbInformation, "FYI..."
```

```
Else
'The OpenTest UDF determines that the workbook is closed.
'A message box asks if you want to open that workbook.
Dim OpenQuestion As Integer
OpenQuestion = _
MsgBox(strFileName & " is not open, do you want to open it?", _
vbYesNo, _
"You choice")

'Example code if you answer No, meaning you want to keep the workbook closed.
If OpenQuestion = vbNo Then
MsgBox "No problem, it'll stay closed.", , "You clicked No."

Else
'Example code if you answer Yes, meaning you want to open the workbook.
'You need to tell the macro what the full path is for this workbook,
'so another String type variable is declared for the path.
Dim strFileFullName As String
strFileFullName = "C:\Your\File\Path\" & strFileName
'Open the workbook.
Workbooks.Open Filename:=strFileFullName
End If

End If

End Sub
```

Adding a Description to the Insert Function Dialog Box

Chances are, the more VBA you learn, the more popular you'll be at your workplace as the Excel go-to person. Soon if not already, you'll be building workbooks for other people to use, and it's a nice touch to add a helpful description to your UDFs for the benefit of those other users. The Insert Function dialog box is a good place to help people understand how to enter your UDFs, especially because this dialog box is how some users enter functions, and each UDF has its own unique entry requirements.

Figure 19-4 shows a typical Insert Function dialog box, where your publicly declared or non-declared UDFs appear in the Select a Function pane when the User Defined category is selected. I've selected the `ExtractNumbers` function, but no help is available for someone who has never seen this UDF and would not know how to properly enter the function.

In two easy steps, here's how you can provide a helpful tip for entering a UDF from the Insert Function dialog box:

1. Press Alt+F8 to call the Macro dialog box. In the Macro Name field, enter the function name; for example, `ExtractNumbers`, as shown in Figure 19-5. Click the Options button.

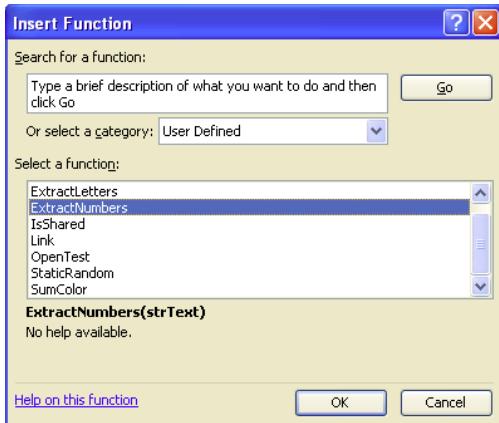


FIGURE 19-4

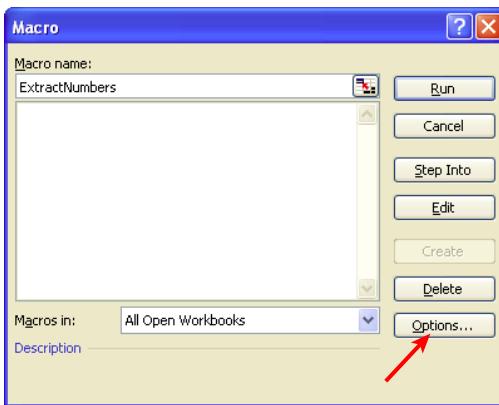


FIGURE 19-5

2. In the Description field of the Macro Options dialog box, enter a brief description of how to enter this UDF. As partially shown in Figure 19-6, I entered `=ExtractNumbers(A2)`, where cell A2 contains the original alphanumeric string. as the description and confirmed it by clicking OK and exiting the Macro dialog box.

And that's all there is to it. Now if you go back to the Insert Function dialog box and select the ExtractNumbers UDF, a description appears, as shown in Figure 19-7, providing the users with a useful tip for how to enter the UDF.

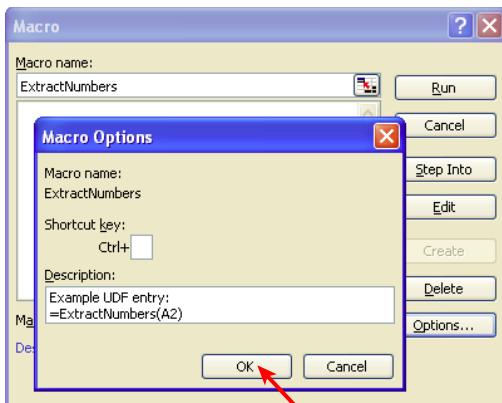


FIGURE 19-6

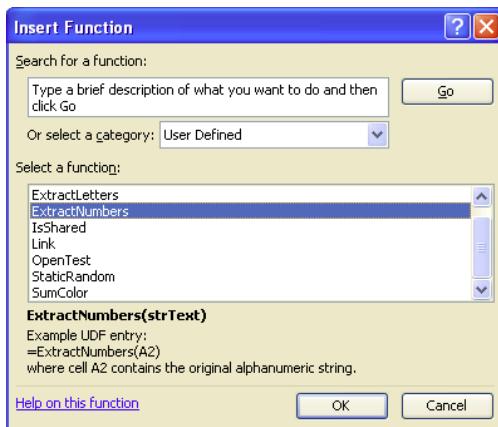


FIGURE 19-7

TRY IT

In this lesson you practice creating a user-defined function that tests whether a particular cell contains a comment. If so, the UDF returns the text of that comment; if not, the UDF returns "No comment".

Lesson Requirements

To get the sample workbook, you can download Lesson 19 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. From your keyboard press Alt+F11 to get into the VBE, and from the menu bar click Insert ➔ Module.
2. Enter the function name, declare an argument variable for a Range type because a cell will be evaluated, and declare the Function type as String because the UDF returns text of some kind. For example:

```
Function GetComment(rng As Range) As String
```

3. Declare a String type variable to handle either the comment text or the "No comment" statement:

```
Dim strText As String
```

4. Using an If structure, evaluate the target cell for the existence of a comment. If there is no comment, define the strText variable as "No comment":

```
If rng.Comment Is Nothing Then  
strText = "No comment"
```

5. Complete the If structure for the condition of the target cell containing a comment:

```
Else  
strText = rng.Comment.Text  
End If
```

6. Set the name of the function equal to the strText string expression:

```
GetComment = strText
```

7. Close the function with the End Function statement. The entire UDF looks like this:

```
Function GetComment(rng As Range) As String  
Dim strText As String  
If rng.Comment Is Nothing Then  
strText = "No comment"  
Else  
strText = rng.Comment.Text  
End If  
GetComment = strText  
End Function
```

8. Press Alt+Q to return to the worksheet. Test your UDF to evaluate the existence of a comment in cell A1 and return the conditional string with this formula in a worksheet cell:

```
=GetComment(A1)
```

REFERENCE Please select the video for Lesson 19 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

20

Debugging Your Code

Despite what you've always heard, there are actually three sure things in life: death, taxes, and errors in computer programs. There's no avoiding it—errors happen and they need to be fixed, whether the length of your VBA programming experience is 10 days or 10 years.

You need to learn the tools and techniques for debugging your code, so that when things go wrong you're familiar with the resources that are at your disposal for finding and fixing errors. Excel has many good built-in debugging tools. In addition, this lesson covers other techniques for avoiding errors in the first place, and, believe it or not, getting errors to work for you instead of against you.

WHAT IS DEBUGGING?

A bug is an error in your code that can produce erroneous results, or, depending on the nature of the bug, stop the code from executing altogether. In programming, the term *debugging* refers to correcting an error in code, or the process of testing a procedure for the possible existence of bugs that would need to be fixed if found.

YOU CAN DO EVERYTHING RIGHT AND STILL HAVE A BUG

The next section covers three causes of errors in VBA programming. Actually, there is a fourth cause, over which you have absolutely no control, and that is a bug in a software application itself. This is not in any way a specific reference to a particular software company or to Microsoft. It's a software-industry reality that new products are sometimes released with bugs, including known bugs that are deemed to be benign but turn out to be a problem when used with Excel.

In your future development projects, you'll encounter many external data storage and management applications that mostly play well with Excel, but sometimes might not when by all rights they should. It's never in any reputable software company's best interests to impose nuisance bugs on its users. The point is, if you find that you have all your bases covered and are still scratching your head about an error that has no rhyme or reason, you might have stumbled onto a bug that other users of that product, and especially the software manufacturer, would want to know about.

The process of debugging is a combination of art and science. The science is covered by some terrific debugging tools that come with Excel VBA. The art is owing to the skills and experience you gain when you build VBA projects with a mindset for anticipating potential minefields based on the intended use—and users—of your projects.

WHAT CAUSES ERRORS?

The world of computer programming enjoys no exemption from Murphy's Law, which states that if something can go wrong, it will go wrong. Three primary types of errors can infect your VBA programming code. To avoid errors, your first line of defense is anticipating problems as you write your code, especially considering how the project will be used in real practice. Eventually, however, one of three types of errors will impose their nuisance selves.

One type is the *syntax error*, such as misspelling a VBA keyword or not declaring a variable while requiring variable declaration (as was outlined in Lesson 6). This causes a compile error as shown in Figure 20-1, because the `LastRow` variable was not declared in that example. If an error can be classified as friendly, it'd be a compile error because it is VBA's way of telling you what's wrong, and sometimes showing you exactly where the problem is.

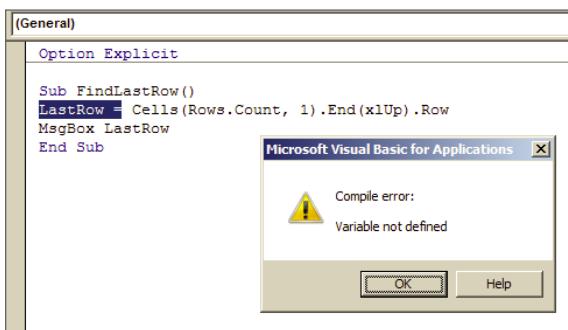


FIGURE 20-1

Another syntax error that can result in a compilation failure is the absence of an `End If`, `End With`, or loop continuation keyword such as `Next` or `Loop`. For example, the macro shown in Figure 20-2 produces a compile error because it is missing an `End With` statement.

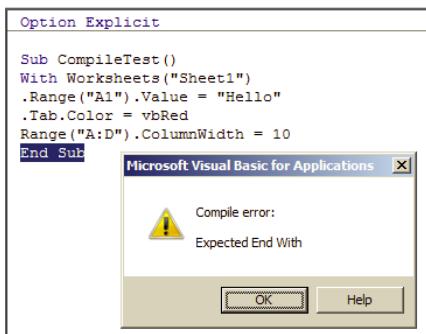


FIGURE 20-2

A second type of error is the *runtime error*, because they occur while the macro is running. These errors usually stop the procedure dead in its tracks with a runtime error message such as the one in Figure 20-3. Notice the reason for the error: In the Project Explorer window, you can see that the workbook has only three worksheets, named Sheet1, Sheet2, and Sheet3.

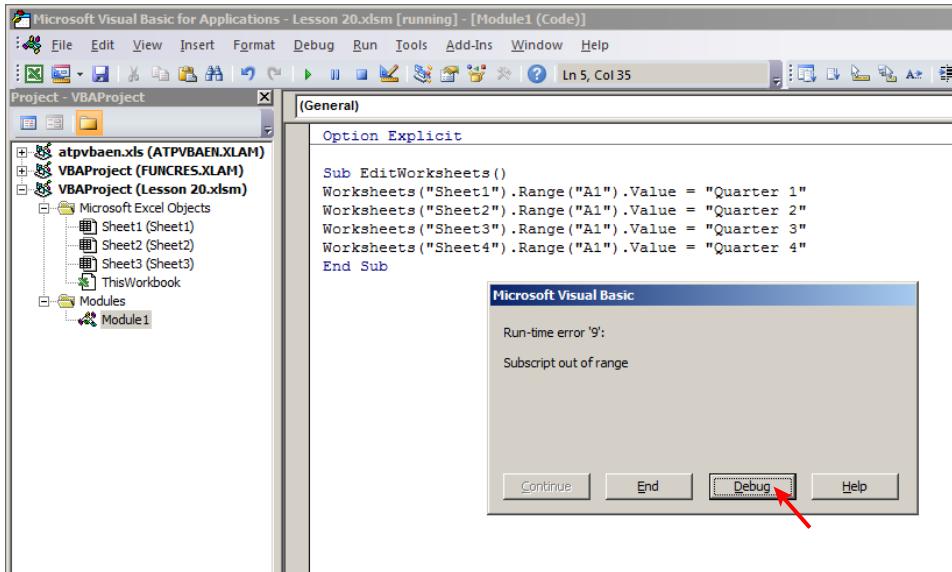


FIGURE 20-3

The runtime error is VBA's way of protesting that it is being told to do something it cannot do, as in this case because a worksheet named Sheet4 does not exist. If the Visual Basic Editor is unprotected, and you click the Debug button on a runtime error message, VBA takes you to the related module and highlights the offending line of code, as shown in Figure 20-4.

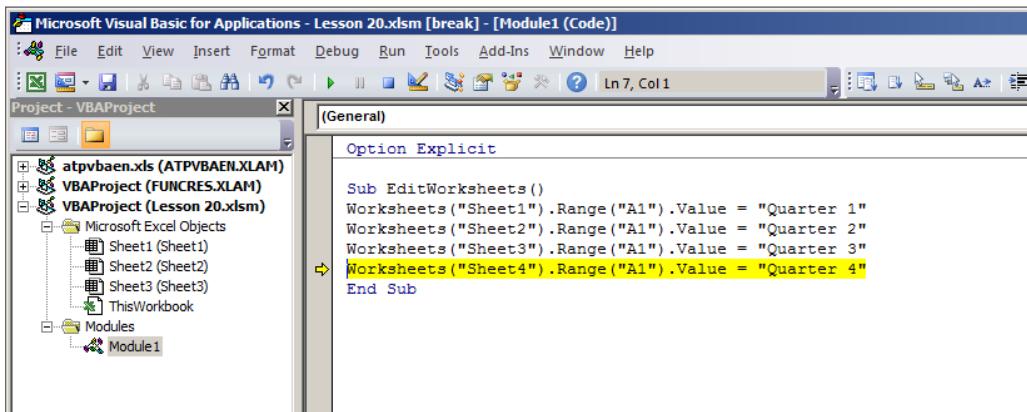


FIGURE 20-4

The third type of error is the *logical error*. These errors are the most nefarious because they come with no message warnings that something is wrong. An example of a logical error is an incorrectly coded

mathematical calculation that yields incorrect results. Suppose your project is a large VBA effort with macros that calculate financial data that end users and investment clients are depending on for their personal investment strategies. Your macros run without getting interrupted by compile or runtime errors, but the results are still flawed. People tend not to fix what they think isn't broken, so unless you (or an angry client) discovers the math bug, it can go undetected for a long time, and it may never be detected.

NOTE When programming mathematical and logical operations, it's always a good idea to test your code by comparing the output of your VBA results with the output from an independent source.

WEAPONS OF MASS DEBUGGING

Now that you know what kinds of bugs are lurking in the shadows and how they can bite your code, you can fight back with several excellent debugging tools that are in the Visual Basic Editor. Your best defense starts with information about the weapons in your debugging arsenal and how they are used.

The Debug Toolbar

The Debug toolbar is a handy item to display and keep docked onto your VBE menu bar. To show the Debug toolbar, from the VBE menu click View \Rightarrow Toolbars \Rightarrow Debug as shown in Figure 20-5.

The Debug toolbar typically contains 13 icons, some of which you are already familiar with. Figure 20-6 shows the toolbar and the names of the icons, and the following sections describe their uses.

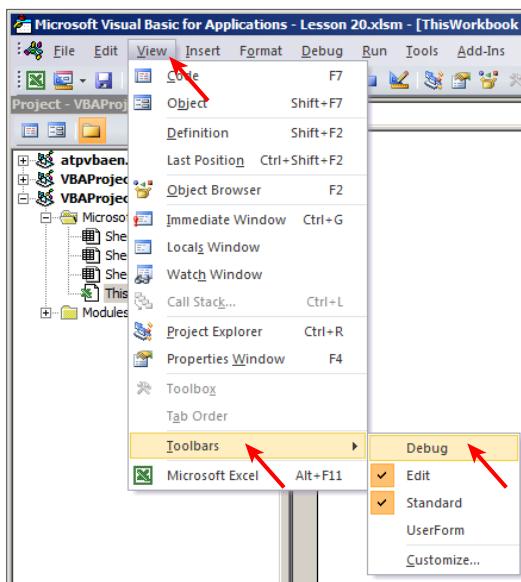


FIGURE 20-5

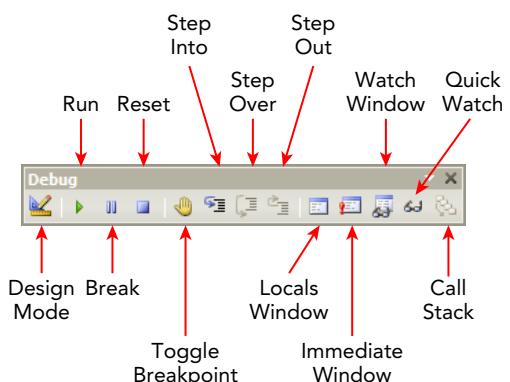


FIGURE 20-6

NOTE There are four VBE toolbars—Debug, Edit, Standard, and UserForm—each of which can remain visible in your VBE by dragging and docking them above or below the menu bar.

Design Mode

The Design Mode button turns Design Mode on and off in the active workbook. Design Mode is the time during which no code from the project is currently running. You can leave Design Mode by clicking the Design Mode icon again, or by running a macro or using the Immediate window. When you have an ActiveX object on your worksheet, such as a `CommandButton`, Design Mode enables you to view the object's properties or to double-click the object to quickly access its module in the VBE.

Run

Clicking the Run button has one of two effects. If your cursor happens to be blinking in the Code window within a macro, clicking the Run button or pressing the F5 key runs the macro. Otherwise, clicking the Run button calls the Macro dialog box, just as if you were on a worksheet and you pressed Alt+F8.

Break

Clicking the Break button is the same as pressing Ctrl+Break, which halts macro execution. Break mode is a special mode of operation in the Visual Basic Editor that enables you to run one line of code at a time without having to run the entire macro. Examining one line of code at a time is a way to pinpoint the exact whereabouts of the error. You can edit code in Break mode.

Reset

Clicking the Reset button clears the call stack and clears the module-level variables. This ends Break mode, ends all program execution, and closes the Debug window if it is open.

Stepping through Code

On the Debug toolbar, three icons—Step Into, Step Over, and Step Out—are related to a process known as *stepping* through code. Sometimes you want to examine each statement in your macro if you suspect a bug is somewhere in your code but you're not sure where. Even large macros can run quickly, so it's difficult, and often impossible, to isolate the specific command that is not executing the way you would have planned. Stepping through your VBA statements enables you to execute one or more lines of code at your own pace to see for yourself what every VBA statement is really doing.

Suppose you oversee a region of 10 hardware stores, and you receive a table of each store's quarterly sales activity. Your table is in a raw form, downloaded into Excel from your company's database, resembling Figure 20-7.

	A	B	C	D	E	F
1						
2						
3						
4						
5	Store 1	23499	19974	3525		
6	Store 2	90970	77324	13646		
7	Store 3	94070	79959	14111		
8	Store 4	29592	25153	4439		
9	Store 5	44800	38080	6720		
10	Store 6	54857	46628	8229		
11	Store 7	77402	65791	11611		
12	Store 8	94547	80364	14183		
13	Store 9	83998	71398	12600		
14	Store 10	62688	53284	9404		
15						

FIGURE 20-7

You have a macro such as the one pictured in Figure 20-8 that formats the table and sorts the Net Income column in descending order so you can quickly list the most profitable stores. When you run the macro, you do not get a compile or runtime error. However, the code did not sort the Net Income in descending order after the macro completed its full execution, as shown in Figure 20-9.

```
Sub StepThroughBreakpointExample()
    'Company name
    Range("A1").Value = "XYZ Widgets, Inc."
    
    'Quarter header label
    Range("A2").Value = "Quarterly Report"
    
    'Call the macro that creates a chart sheet from this data.
    Call myChartMaker
    
    'Enter the table headers and bold them.
    With Range("B4:E4")
        .Value = Array("Store Name", "Gross Sales", "Expenses", "Net Income")
        .Font.Bold = True
    End With
    
    'Border around the table.
    Range("B4:E14").BorderAround Weight:=xlMedium
    
    'Format the numbers to currency and comma separators.
    Range("C5:E14").NumberFormat = "$#,##0"
    
    'AutoFit the columns from A:E
    Range(Columns(1), Columns(5)).AutoFit
    
    'Sort the table by descending Net Income column E.
    With ActiveSheet.Sort
        .SortFields.Clear
        .SortFields.Add Key:=Range("E5:E13"),
        SortOn:=xlSortOnValues, Order:=xlDescending
        .SetRange Range("B5:E13")
        .Header = xlYes
        .Apply
    End With
End Sub
```

FIGURE 20-8

A	B	C	D	E	F
1	XYZ Widgets, Inc.				
2	Quarterly Report				
3					
4	Store Name	Gross Sales	Expenses	Net Income	
5	Store 1	\$23,499	\$19,974	\$3,525	
6	Store 8	\$94,547	\$80,364	\$14,183	
7	Store 3	\$94,070	\$79,959	\$14,111	
8	Store 2	\$90,970	\$77,324	\$13,646	
9	Store 9	\$83,998	\$71,398	\$12,600	
10	Store 7	\$77,402	\$65,791	\$11,611	
11	Store 6	\$54,857	\$46,628	\$8,229	
12	Store 5	\$44,800	\$38,080	\$6,720	
13	Store 4	\$29,592	\$25,153	\$4,439	
14	Store 10	\$62,688	\$53,284	\$9,404	
15					

FIGURE 20-9

Using the Step Into Command

To examine line by line where the problem lies, click your mouse anywhere inside the macro and then click the Step Into button. The macro's Sub line is highlighted in yellow, indicating to you that it's that particular macro you are about to step into.

NOTE When you “step into” a macro, you are traversing step-by-step (code line by code line), in a single-step process to execute each line in turn.

Click the Step Into button again and the first line of code—which in this example is `Range("A1").Value = "XYZ Widgets, Inc."`—is highlighted in yellow, as shown in Figure 20-10. If you click the Step Into button again, the code line `Range("A1").Value = "XYZ Widgets, Inc."` is executed, and the next line of code—`Range("A2").Value = "Quarterly Report"`—is highlighted in yellow, ready to be executed with your next Step Into command.

Each time you click the Step Into button, the line of code that is highlighted is executed, and the next line is highlighted, and so on until you reach the end of the macro. Because you suspect a bug somewhere in the code, you'd be looking at your worksheet after each Step Into command to make sure that what the code is supposed to be doing is what it truly is doing.

In this example, all the cell values and formatting were correctly executed when you stepped into each one, until the very last section of code that executes the `Sort` method. You find when stepping into that section that the range of cells being sorted is not correct. Your table occupies range B4:E14 but the VBA code is sorting only up to row 13. Your suspicions were correct about the final result on the worksheet looking peculiar, so you make a quick adjustment to the sort range address after you've verified that each of the other lines of code were properly written and being properly executed.

```

Sub StepThroughBreakpointExample()

    'Company name
    Range("A1").Value = "XYZ Widgets, Inc."

    'Quarter header label
    Range("A2").Value = "Quarterly Report"

    'Call the macro that creates a chart sheet from this data.
    Call myChartMaker

    'Enter the table headers and bold them.
    With Range("B4:E4")
        .Value = Array("Store Name", "Gross Sales", "Expenses", "Net Income")
        .Font.Bold = True
    End With

    'Border around the table.
    Range("B4:E14").BorderAround Weight:=xlMedium

    'Format the numbers to currency and comma separators.
    Range("C5:E14").NumberFormat = "$#,##0"

    'AutoFit the columns from A:E
    Range(Columns(1), Columns(5)).AutoFit

    'Sort the table by descending Net Income column E.
    With ActiveSheet.Sort
        .SortFields.Clear
        .SortFields.Add Key:=Range("E5:E13"), _
        SortOn:=xlSortOnValues, Order:=xlDescending
        .SetRange Range("B5:E13")
        .Header = xlYes
        .Apply
    End With

End Sub

```

FIGURE 20-10

Using the Step Over Command

The Step Over command is similar to the Step Into command, with the difference between the two commands occurring at the point of a call to another macro. You may have noticed in the macro the code line `Call myChartMaker`, where in this hypothetical example the `myChartMaker` macro creates a chart sheet from the table data. Figure 20-11 shows that `Call` statement highlighted during the Step Into process.

In this situation, if you click the Step Over button, the `Call myChartMaker` command is executed but you are not taken through it line by line as if it were stepped into. You would prefer to do this when you know for sure that the `myChartMaker` macro works without any problems and cannot be the cause of whatever bug you are trying to fix in the current macro. The Step Over command executes the `myChartMaker` macro, and the next line of code in your macro is highlighted for the next Step Into command.

NOTE Did you notice a tiny arrow in the margin to the left of the macro being stepped into? When a line of code is highlighted during a stepping process, a yellow arrow in the Code window's left margin helps to indicate your place in the process. With your mouse, you can select and drag the arrow upward or downward, dropping it at whichever line of code you want to execute next.

```

Sub StepThroughBreakpointExample()

    'Company name
    Range("A1").Value = "XYZ Widgets, Inc."

    'Quarter header label
    Range("A2").Value = "Quarterly Report"

    'Call the macro that creates a chart sheet from this data.
    Call myChartMaker

    'Enter the table headers and bold them.
    With Range("B4:E4")
        .Value = Array("Store Name", "Gross Sales", "Expenses", "Net Income")
        .Font.Bold = True
    End With

    'Border around the table.
    Range("B4:E14").BorderAround Weight:=xlMedium

    'Format the numbers to currency and comma separators.
    Range("C5:E14").NumberFormat = "$#,##0"

    'Autofit the columns from A:E
    Range(Columns(1), Columns(5)).AutoFit

    'Sort the table by descending Net Income column E.
    With ActiveSheet.Sort
        .SortFields.Clear
        .SortFields.Add Key:=Range("E5:E13"), _
            SortOn:=xlSortOnValues, Order:=xlDescending
        .SetRange Range("B5:E13")
        .Header = xlYes
        .Apply
    End With

End Sub

```



FIGURE 20-11

Using the Step Out Command

The Step Out command executes the remaining lines of code between and including the current highlighted execution point and the `End Sub` line. You might think by the name Step Out that it refers to simply exiting the Step Into command, but that is not exactly the case. Though it does result in exiting the step-through process, it does so by executing the rest of the macro to get to the end. If you want to exit any of the step-through process, click the Reset button.

Toggle Breakpoint

One of VBA's convenient features is the ability to set a *breakpoint*, where you can specify a line of code that is the point up to which the macro will run at full speed. When the macro's execution reaches the breakpoint code line, VBA switches to Break mode and halts the execution process.

NOTE Stepping through your macro is a good way to examine each line of code, but when your macros are hundreds of lines long, a line-by-line examination process is tedious and time-consuming. There will be many statements in your code that won't need to be examined, and there's no reason to inch your way to the section of your macro where the error probably resides. This is where breakpoints come in handy.

To set a breakpoint in your code, click your mouse into the line of code where you want the breakpoint to start. Click the Toggle Breakpoint button or press the F9 key, and the breakpoint is set at that line. VBA clearly identifies a breakpoint with a large brown dot in the Code window's left margin, and the code line itself is shaded brown.

For example, if you suspect a bug in a macro but you know that the majority of the macro runs without any problems, you can set a breakpoint starting at a section in the program where you want to examine the code more closely. In Figure 20-12, I clicked my mouse into the code line `With ActiveSheet.Sort` and clicked the Toggle Breakpoint button. If the macro were to be run now, it would execute all lines of code up to, but not including, that breakpoint line. Now, you can step through the subsequent lines of code to verify that each line is doing what you'd expect.

```
Sub StepThroughBreakpointExample()

    'Company name
    Range("A1").Value = "XYZ Widgets, Inc."

    'Quarter header label
    Range("A2").Value = "Quarterly Report"

    'Call the macro that creates a chart sheet from this data.
    Call myChartMaker

    'Enter the table headers and bold them.
    With Range("B4:E4")
        .Value = Array("Store Name", "Gross Sales", "Expenses", "Net Income")
        .Font.Bold = True
    End With

    'Border around the table.
    Range("B4:E14").BorderAround Weight:=xlMedium

    'Format the numbers to currency and comma separators.
    Range("C5:E14").NumberFormat = "$#,##0"

    'AutoFit the columns from A:E
    Range(Columns(1), Columns(5)).AutoFit

    'Sort the table by descending Net Income column E.
    With ActiveSheet.Sort
        .SortFields.Clear
        .SortFields.Add Key:=Range("E5:E13"), _
        SortOn:=xlSortOnValues, Order:=xlDescending
        .SetRange Range("B5:E13")
        .Header = xlYes
        .Apply
    End With

End Sub
```

FIGURE 20-12

NOTE You can set a breakpoint only on an executable line. Commented lines in your code, or empty lines, cannot be set as breakpoints.

True to its name, you can click the Toggle Breakpoint button again to clear the current breakpoint with any portion of that line selected, or you can click the large dot in the Code window's margin.

If you have already set a breakpoint and you click the Toggle Breakpoint button or press F9, you set another breakpoint if you have any other line of code selected. You can set more than one breakpoint, so to quickly clear all breakpoints at once, press Ctrl+Shift+F9.

Locals Window

The Locals window can help you in situations in which you get a runtime error and the offending line of code involves a variable. The Locals window displays the variables and their values for the macro(s) you are currently running.

Figure 20-13 shows a very simple macro that attempted to activate a worksheet based on the object variable `mySheet`. Because that variable was never set with an identifying worksheet, a runtime error occurred because VBA could not determine which sheet the `mySheet` variable was referring to. While in Break mode in this example, the Locals window shows that `mySheet` is set to `Nothing`, telling you that you forgot to include a `Set` statement for `mySheet`.

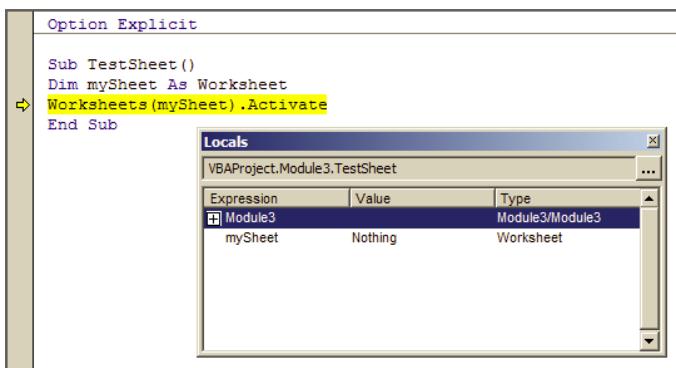


FIGURE 20-13

Immediate Window

The Immediate window enables you to type in or paste a line of VBA code, which executes when you press the Enter key. To see the Immediate window, you can click its icon on the Debug toolbar, select View \Rightarrow Immediate Window, or press Ctrl+G.

If it hasn't happened already, you'll soon find yourself using the Immediate window for reasons having nothing to do with errors. The Immediate window is a great way to execute commands quickly without needing to create a formal macro to get the task done, such as in the following examples.

To eliminate leading apostrophes in cell values, which can occur when manually entered or imported from external source data, you can type `Activesheet.UsedRange.Value = Activesheet.UsedRange.Value` and press Enter. To delete hyperlinks but keep the underlying cell value, you can type `ActiveSheet.Hyperlinks.Delete` and press Enter.

When querying some fact or condition, precede your statement with a leading question mark. If you want to know the version of Excel you are using, type `? Application.Version` and press the Enter key. As shown in Figure 20-14, when I entered that statement into the Immediate window, the value 15.0 was returned, which is Excel's version 2013.

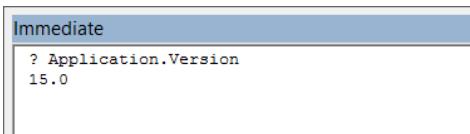


FIGURE 20-14

The point to be made about the Immediate window is that it is a proactive tool. If you are wondering whether a line of code will fail, or whether it will produce the result you have envisioned, you can test that code line in the Immediate window and see the results before taking your chances and putting it into your code.

Watch Window

The Watch window enables you to watch a variable or an expression change as your code executes. You'd normally do this with values that are associated with runtime errors, so you can see at what point the VBA expressions produced a value that might have caused the error.

Select the expression you want to watch, right-click that selection, and choose Add Watch from the pop-up menu. Figure 20-15 shows the process for adding the variable `strValue` to the Watch list. The Add Watch dialog box displays, as shown in Figure 20-16, for you to confirm your settings and click OK.

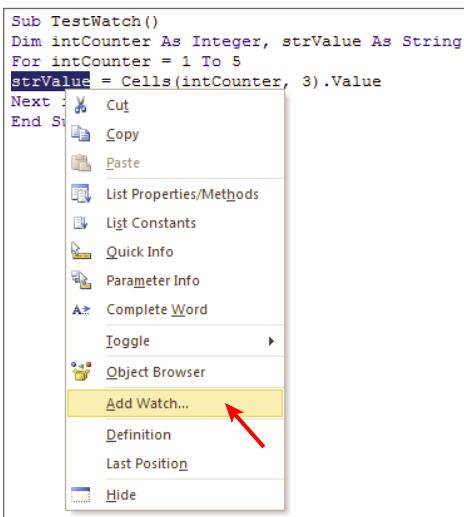


FIGURE 20-15

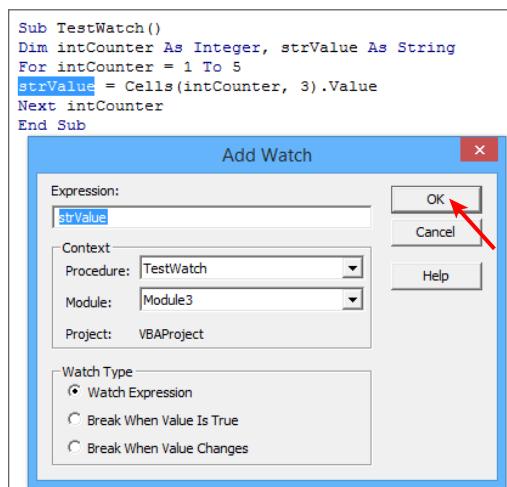


FIGURE 20-16

When you step into code after setting a watch expression, you see the expression's value change during execution. Figure 20-17 shows the `strValue` variable's value change with each iteration of the `For Next` loop. Notice that the value at one point in the loop is a number, yet the `strValue` variable was declared as a `String` type. It's that kind of attention that the Watch window brings to your awareness of what your variables are actually returning, if you suspect a particular expression to be the cause of an error.

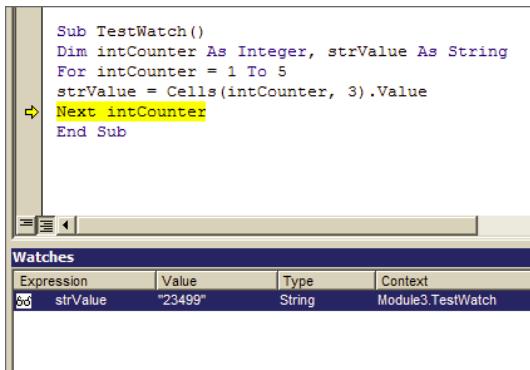


FIGURE 20-17

Quick Watch

The Quick Watch window enables you to get a look at the current value of an expression or variable for which you have not defined a watch expression. While you are in Break mode, select your expression in the module and click the Quick Watch button, or press Shift+F9. For example, in Figure 20-18, the intCounter variable was selected during a step-through process, and the Watch window displays 3 in the Value field, indicating that the For Next loop is currently in its third iteration.

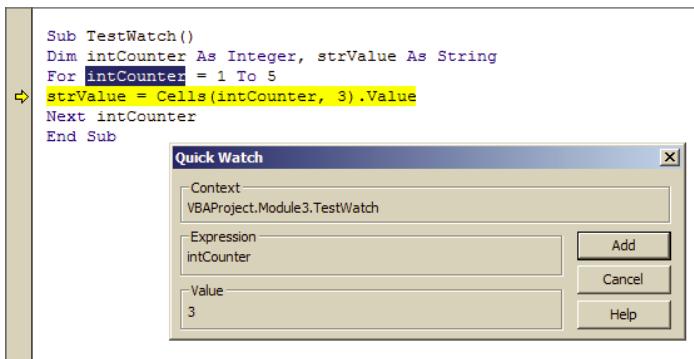


FIGURE 20-18

Call Stack

The Call Stack dialog box shows the list of currently active procedure calls in Break mode. Unless you write macros that involve a maze of calls to other macros, that themselves call other procedures, you won't need the Call Stack dialog box. A word to the wise: Keep your macros simple and limit their procedure calls to a reasonable level, and you won't have to worry about relying on a dialog box to tell you which macro is in error in Break mode.

TRAPPING ERRORS

When you encounter a runtime error and you've figured out the cause, it might be that you need to keep the error-prone code in place because it is such an important component of the larger macro. Actually, you will come across this situation a lot, so you'll need to know how to handle errors programmatically behind the scenes, in a way that the users of your projects will not be bothered by runtime errors.

Error Handler

One of the more common tasks in development projects is to add a worksheet to the workbook. Your project might involve building a report onto a new worksheet, or copying various sections of a master worksheet and pasting those individual sections to their own new worksheets that you create. Say you provide an InputBox for users to enter the name of a worksheet they want to add. What happens if a user already has a worksheet by that name in the workbook? Two worksheets cannot have the same name in the same workbook, but the macro still needs to complete its appointed task.

One approach is using an `On Error GoTo` statement that traps the error and points to a certain section in your macro that should be executed next to handle the error. Suppose your macro calls for a new worksheet to be added and named by the user as `Sheet3`. If a worksheet already exists in the workbook named `Sheet3`, a 1004 type runtime error message would occur as shown in Figure 20-19.

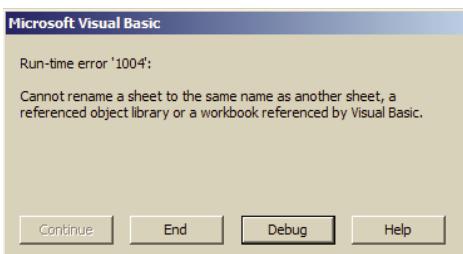


FIGURE 20-19

With the following syntax, you can use an error handler to avoid getting a runtime error message if an attempt is made to give a new worksheet the same name another worksheet already has. In this example macro, the user is provided an InputBox to name the new sheet, and informed if the sheet is added, or if it is not added because duplicate names are not allowed:

```
Sub AddSheetTest()
    Dim mySheetName As String
    mySheetName = _
        InputBox("Enter the worksheet name:", _
            "Add and name a new worksheet")
    If mySheetName = "" Then Exit Sub
    On Error GoTo ErrorHandler
    Worksheets.Add.Name = mySheetName
    MsgBox _
        "Worksheet " & mySheetName & " was added.", , "Thank you."
    Exit Sub
ErrorHandler:
    MsgBox _
```

```

"A worksheet named " & mySheetName & " already exists.", _
vbCritical, _
"Duplicate sheet names are not allowed."
End Sub

```

Bypassing Errors

My preference for most situations where runtime errors can occur is to avoid the error handler route because the `GoTo` statement makes the macro more difficult to follow. Using an error-bypass approach with the `On Error Resume Next` statement, you can test for the condition of the `Error` object and use an `If` structure to deal with either possibility.

When it comes to naming a sheet, you need to monitor several considerations:

- Does the sheet name already exist in the workbook? Duplicate sheet names are not allowed.
- Is the proposed sheet name more than the maximum allowable 31 characters in length?
- Are any illegal sheet-naming characters included in the proposed name? Sheet tab names cannot contain the characters /, \, [,], *, ?, or :. If you try to type any of those characters into your sheet tab, Excel disallows the entry.

The following macro takes these possibilities into consideration. If all conditions are met, a new sheet is added. If any condition is not met, a new worksheet is not created, and a message box informs you of the reason why:

```

Sub TestSheetCreate()

'Declare String type variables for naming and testing the sheet.
Dim mySheetName As String, mySheetNameTest As String

'Use an InputBox to ask the user to propose a new sheet name.
mySheetName = _
InputBox("Enter the worksheet name:", _
"Add and name a new worksheet")

'Exit if nothing was entered or the Cancel button was clicked.
If mySheetName = "" Then Exit Sub

'Error bypass if the proposed sheet name already exists
'in the workbook.
On Error Resume Next
mySheetNameTest = Worksheets(mySheetName).Name
If Err.Number = 0 Then
MsgBox _
"The sheet named " & mySheetName & " already exists.", _
vbInformation, _
"A new sheet was not added."
Exit Sub
End If

'If the length of the proposed sheet name exceeds 31 characters,
'disallow the attempt.
If Len(mySheetName) > 31 Then
MsgBox _
"Worksheet tab names cannot exceed 31 characters." & vbCrLf & _

```

```
"You entered " & mySheetName & ", which has " & vbCrLf & _
Len(mySheetName) & " characters.", vbInformation, _
"Please use no more than 31 characters."
Exit Sub
End If
'Sheet tab names cannot contain
'the characters /, \, [ , ], *, ?, or :.
'Verify that none of these characters
'are present in the cell's entry.
Dim IllegalCharacter(1 To 7) As String, i As Integer
IllegalCharacter(1) = "/"
IllegalCharacter(2) = "\"
IllegalCharacter(3) = "["
IllegalCharacter(4) = "]"
IllegalCharacter(5) = "*"
IllegalCharacter(6) = "?"
IllegalCharacter(7) = ":".
'Loop through each character in the proposed sheet name.
For i = 1 To 7
If InStr(mySheetName, (IllegalCharacter(i))) > 0 Then
MsgBox _
"You included a character that Excel does not allow" & vbCrLf & _
"when naming a sheet. Please re-enter a sheet name" & vbCrLf & _
"without the '" & IllegalCharacter(i) & "' character.", _
vbCritical, _
"Sheet not added."
Exit Sub
End If
Next i
'History is a reserved word, so a sheet cannot be named History.
If UCase(mySheetName) = "HISTORY" Then
MsgBox "A sheet cannot be named " & mySheetName & vbCrLf & _
"because it is a reserved word in Excel.", vbInformation, _
"History is a reserved word."
Exit Sub
End If

'Inform the user that a new sheet has been added.
Worksheets.Add.Name = mySheetName
MsgBox "A new sheet named " & mySheetName & " has been added!", _
vbInformation, _
"Thank you!"
End Sub
```

TRY IT

In this lesson, you create a macro that avoids a runtime error while using the `Find` method to locate a value on your worksheet. If the value is found, its cell address is displayed in a message box.

If you were to record a macro to find the word Hello on a worksheet, the recorded code would look like this:

```
Cells.Find(What:="Hello", After:=ActiveCell, LookIn:=xlFormulas, _
LookAt:=xlPart, SearchOrder:=xlByRows, SearchDirection:=xlNext, _
MatchCase:=False, SearchFormat:=False).Activate
```

If the word Hello is not found on the worksheet, a runtime error would result because the recorded code is instructing VBA to activate a cell that contains a value that does not exist. The purpose of this lesson is to avoid a runtime error if the value being looked for does not exist on the worksheet.

Lesson Requirements

To get the sample workbook, you can download Lesson 20 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

It is not practical to loop through potentially millions of cells, so you use the `Find` method with an error bypass structure.

Step-by-Step

1. Open a workbook and activate a worksheet that contains a relatively large amount of data. This is an exercise in finding a value if it exists on the worksheet, so the more complex the worksheet, the better.
2. From your worksheet press Alt+F11 to get into the Visual Basic Editor.
3. From the menu bar, click Insert \Rightarrow Module.
4. In your new module, type the name of your macro as `Sub FindTest` and press Enter. VBA displays your entry and new macro as follows:

```
Sub FindTest()
```

```
End Sub
```

5. For your first line of code, declare a Variant type variable for the value you want to locate. In this example, simply call it `varFind`:

```
Dim varFind as Variant
```

6. Declare a String type variable for the value to be located:

```
Dim FindWhat As String
```

7. Define the `FindWhat` variable as an InputBox entry:

```
FindWhat =  
InputBox("What do you want to find?", "Find what?")
```

8. If the Cancel button is clicked, or nothing is entered in the InputBox, exit the macro:

```
If FindWhat = "" Then Exit Sub
```

9. Set the `varFind` variable to the `Find` method:

```
Set varFind =  
Cells.Find(What:=FindWhat, LookIn:=xlFormulas, lookat:=xlWhole)
```

10. If `varFind` is Nothing, inform the user that the value being looked for was not found. Also, exit the macro:

```
If varFind Is Nothing Then  
MsgBox _
```

```

FindWhat& " was not found.", _
vbInformation, _
"No such animal."
Exit Sub
Else

```

11. A message box informs the user that the value was found, and in what cell:

```
MsgBox FindWhat& " was found in cell " & varFind.Address, , "Found"
```

12. Enter the End If statement:

```
End If
```

13. Press Alt+Q to return to the worksheet and test your macro. The entire macro when it is completed looks like this:

```

Sub FindTest()

'Declare a variant type variable for the value to locate.
Dim varFind As Variant
Dim FindWhat As String

'Define the FindWhat variable as an InputBox entry.
FindWhat = _
InputBox("What do you want to find?", "Find what?")

'If the Cancel button is clicked, or nothing is entered
'in the InputBox, exit the macro.
If FindWhat = "" Then Exit Sub

'Set the varFind variable to the Find method.
Set varFind = _
Cells.Find(What:=FindWhat, LookIn:=xlFormulas, lookat:=xlWhole)

'If varFind = Nothing, inform the user that the value being
'looked for was not found. Also, exit the macro.
If varFind Is Nothing Then
MsgBox _
FindWhat& " was not found.", _
vbInformation, _
"No such animal."
Exit Sub
Else

'A message box informs the user that the value was found,
'and in what cell.
MsgBox FindWhat& " was found in cell " & varFind.Address, , "Found"
End If
End Sub

```

REFERENCE Please select the video for Lesson 20 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

PART IV

Advanced Programming Techniques

- ▶ **LESSON 21:** Creating UserForms
- ▶ **LESSON 22:** UserForm Controls and Their Functions
- ▶ **LESSON 23:** Advanced UserForms
- ▶ **LESSON 24:** Class Modules
- ▶ **LESSON 25:** Add-Ins
- ▶ **LESSON 26:** Managing External Data
- ▶ **LESSON 27:** Data Access with ActiveX Data Objects
- ▶ **LESSON 28:** Impressing Your Boss (or at Least Your Friends)

21

Creating UserForms

In previous lessons, you have seen examples of how your workbook can interact with its users to make decisions by employing such methods as InputBoxes and Message Boxes. Although these interactive tools are very useful for the situations they are meant to serve, they have limited usefulness in more complex applications.

Some of your projects will require a more versatile approach to asking for and gathering many kinds of information from users, all within a dedicated interface that's convenient and easy to use. Perhaps you have seen attempts to accomplish this on a neatly arranged worksheet where certain cells are color-shaded or unprotected for data input, maybe with drop-down lists and embedded check boxes or option buttons. A UserForm in VBA is a more efficient method for collecting and recording such information.

WHAT IS A USERFORM?

A *UserForm* is essentially a custom-built dialog box, but that description does not do justice to the immense complexity and diversity with which UserForms can be built and be made to function. A UserForm is created in the Visual Basic Editor, with controls and associated VBA code, usually meant for the end user to be advised of some information or to enter data, generate reports, or perform some action.

NOTE *Think of UserForms as electronic versions of the different forms you fill out on your computer, such as when you make an online purchase, or with paper and pen in a business office. Some information on most forms is required and some information is optional. A UserForm is a dynamic object, with VBA code working behind the scenes to guide your users toward telling your workbook what it needs to know.*

CREATING A USERFORM

The first step in creating a new UserForm is to insert one into the Visual Basic Editor. To do that, press Alt+F11 to get into the VBE, and select your workbook name in the Project Explorer as shown in Figure 21-1.

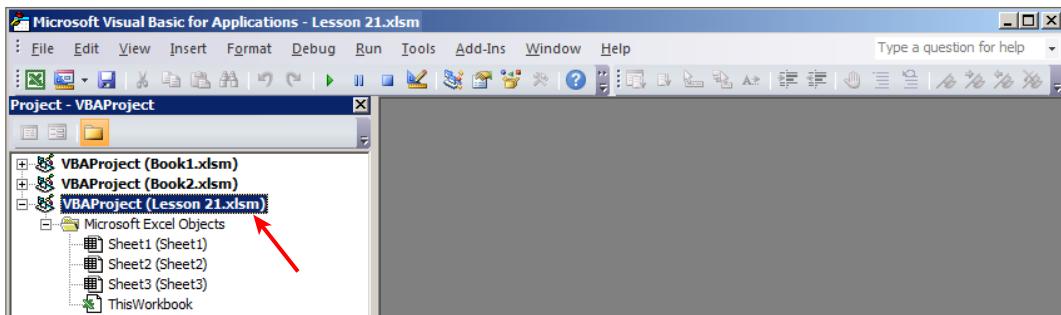


FIGURE 21-1

NOTE Be careful to select the workbook you have in mind before adding a UserForm to it! In Figure 21-1, a couple other workbooks are open to help make the point that the workbook of interest (Lesson21.xlsxm in this example) is the workbook selected in the Project Explorer.

With the workbook name selected, from the menu bar click Insert \Rightarrow UserForm as shown in Figure 21-2.

A new UserForm opens in its design window as shown in Figure 21-3.

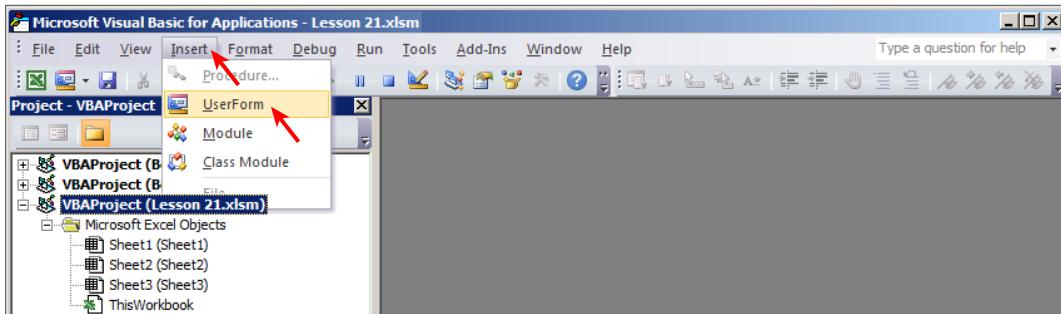


FIGURE 21-2

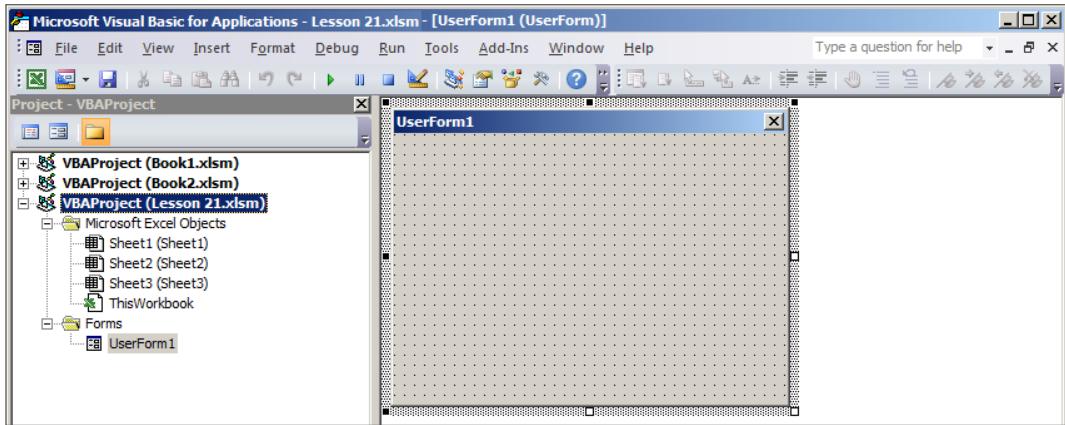


FIGURE 21-3

DESIGNING A USERFORM

UserForms have a variety of properties. You can show the Properties window for the UserForm itself, or for any of its controls, by selecting the object and clicking its Properties icon, or clicking View \Rightarrow Properties Window as shown in Figure 21-4.

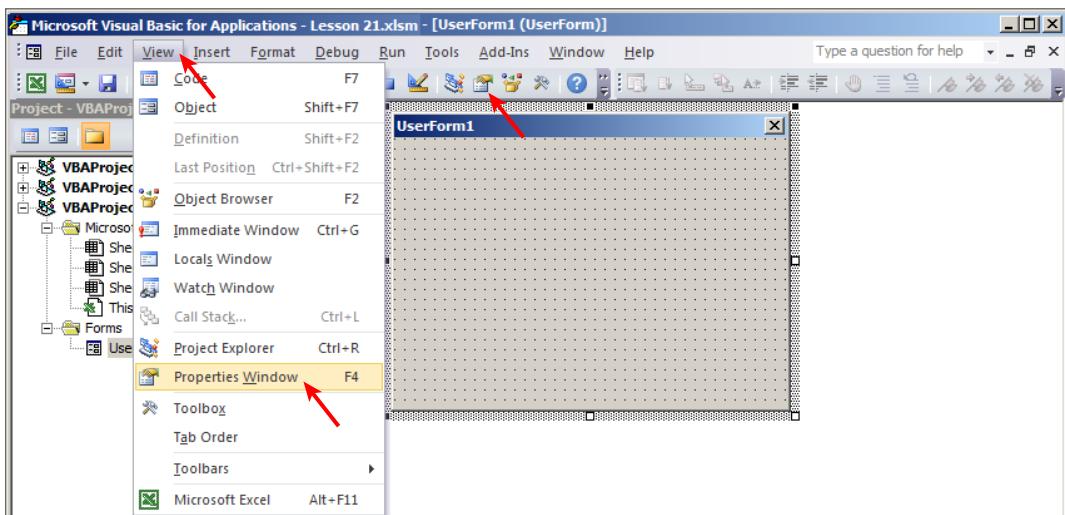


FIGURE 21-4

Below the Project Explorer is where you'll see the Properties window, partially visible in Figure 21-5.

For the workbook's first UserForm, VBA assigns a default value of UserForm1 to its Name and Caption properties, as you can see in Figure 21-5. If you were to create a second UserForm, its default Name and Caption properties would be UserForm2, and so on. To help distinguish between the Name and Caption properties, Figure 21-6 shows where the Name property has been changed to frmEmployees, and the Caption property, which is displayed in the UserForm's title bar, has been changed to Employee Information.

NOTE When naming UserForms, or any object for that matter, it's best to assign a name that is relevant to the theme of the object. When I name a UserForm, I use the prefix `frm` (for UserForm) followed by a simple, intuitive term (such as `Employees` in this example) that represents the basic idea of the UserForm object.

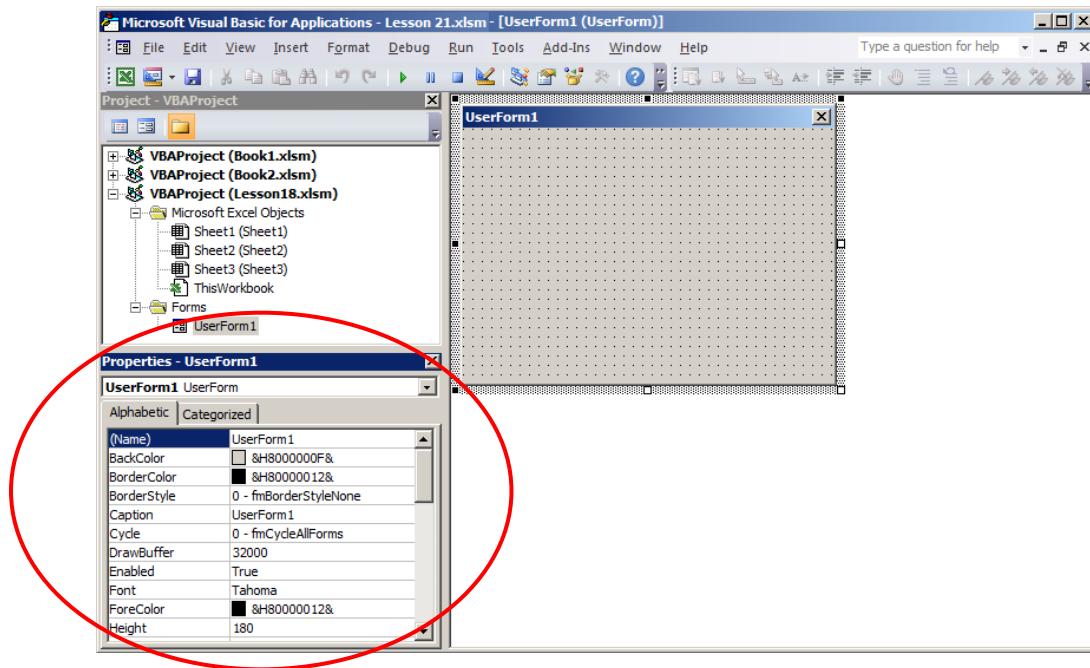


FIGURE 21-5

ADDING CONTROLS TO A USERFORM

A *control* is an object such as a Label, TextBox, OptionButton, or CheckBox in a UserForm or embedded onto a worksheet that allows users to view or manipulate information. VBA supports these and more controls, which are accessible to you from the VBE Toolbox. To show the Toolbox

so you can easily grab whatever controls you want from it, you can click the Toolbox icon, or click View ➔ Toolbox as shown in Figure 21-7.

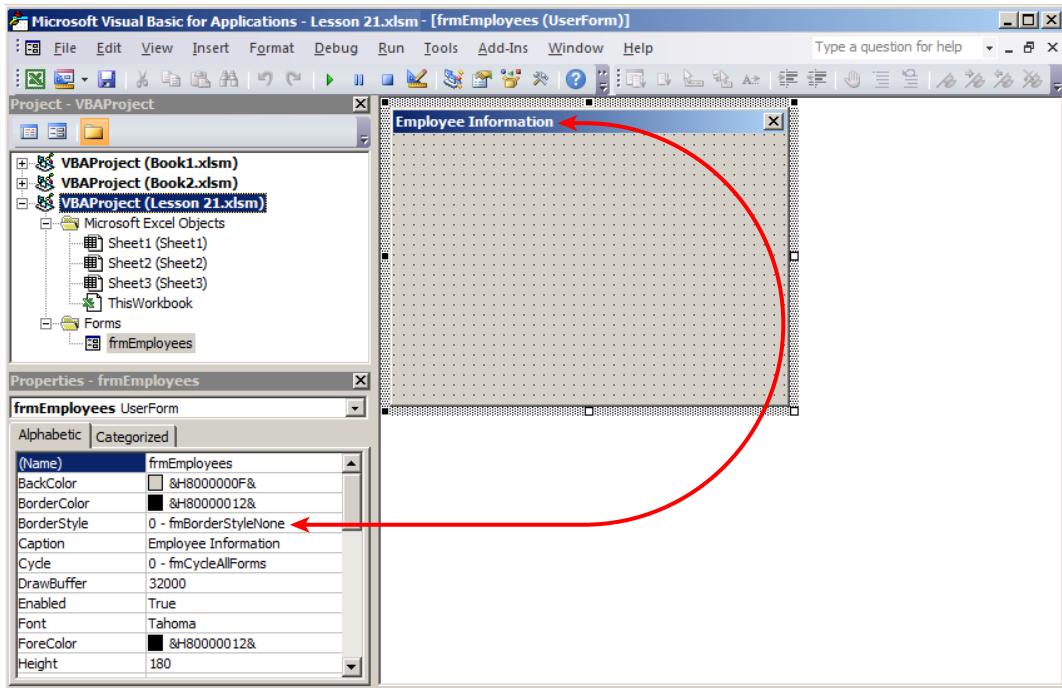


FIGURE 21-6

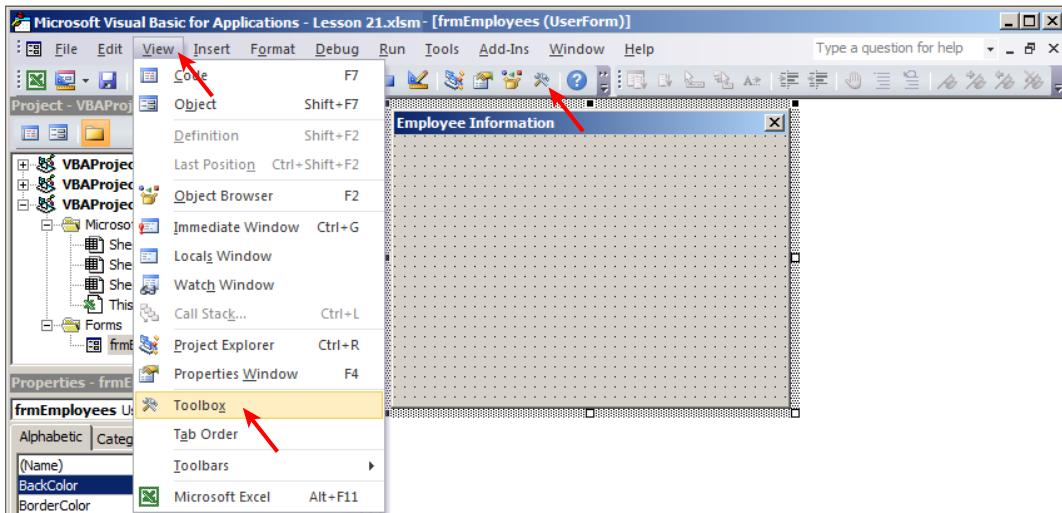


FIGURE 21-7

The control(s) you place onto your UserForm depend on its purpose. If you want to design a simple form to gather employee information for your company, you'd at least want to know the employees' names and their titles. It would be useful to display a TextBox to enter the employee name, and then a list of the company's position titles so the user can effortlessly select one. Figure 21-8 shows the Toolbox with the mouse hovering over the Label control icon.

You place a control onto your UserForm by drawing the control onto your UserForm's design area. All you need to do is click whatever Toolbox control icon you're interested in adding to the UserForm, and draw it as you would draw a Shape object onto a worksheet. Figure 21-9 shows a Label control that was just drawn, showing its default caption of Label1.

Notice in Figure 21-9 that the Label's Caption property is selected in the Properties window, so a more meaningful caption can be added to the Label. Because the Label will be directly above the TextBox, and the purpose of the TextBox is to enter an employee name, the Label's caption is changed to Employee Name as shown in Figure 21-10. Notice further in Figure 21-10 that the TextBox icon is about to be selected in the Toolbox, as you get ready to draw a TextBox control onto the UserForm below the Label.

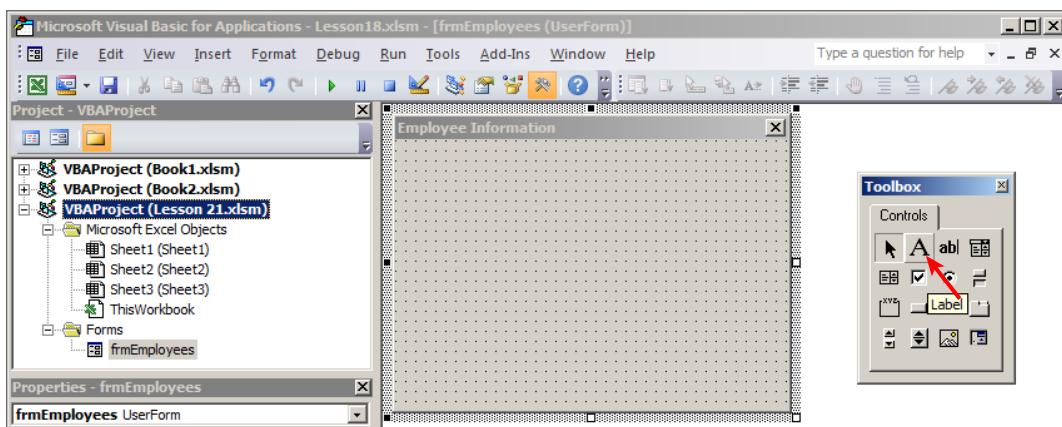


FIGURE 21-8

After you click the Toolbox's TextBox icon, you add a TextBox control by drawing it onto the UserForm's design area, just as you did when you added the Label control. Figure 21-11 shows the drawn TextBox, positioned below the Label, and having a reasonably sufficient width to accept and display a person's name. Meanwhile, as you can see in Figure 21-11, the Frame icon is about to be selected in preparation for placing a Frame control onto your UserForm.

Figure 21-12 shows your just-drawn Frame control with its default caption of Frame1. Frames are a good way to group other controls visually by containment, usually with an underlying theme. In the case of this UserForm example, the company's position titles will be contained in such a way that the user can select only one.

The caption of a Frame control is an efficient way to describe the purpose of the Frame, just as the Label's caption of Employee Name describes the purpose of the TextBox. In Figure 21-12, the Caption property of your new Frame is selected so you can change the meaningless default caption of Frame1 to a more useful description.

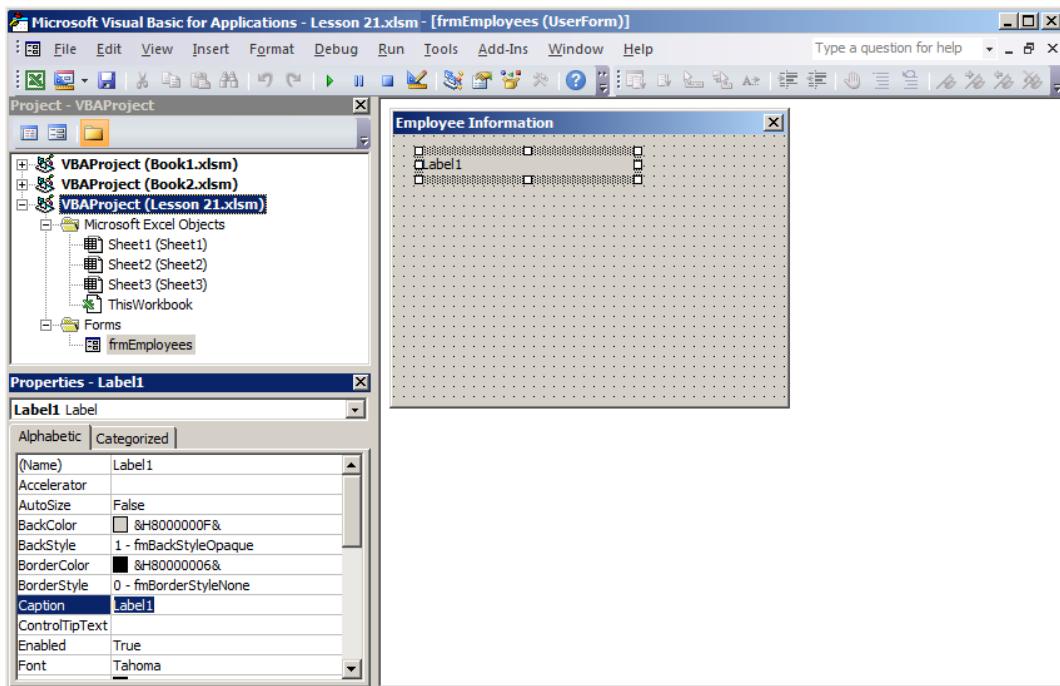


FIGURE 21-9

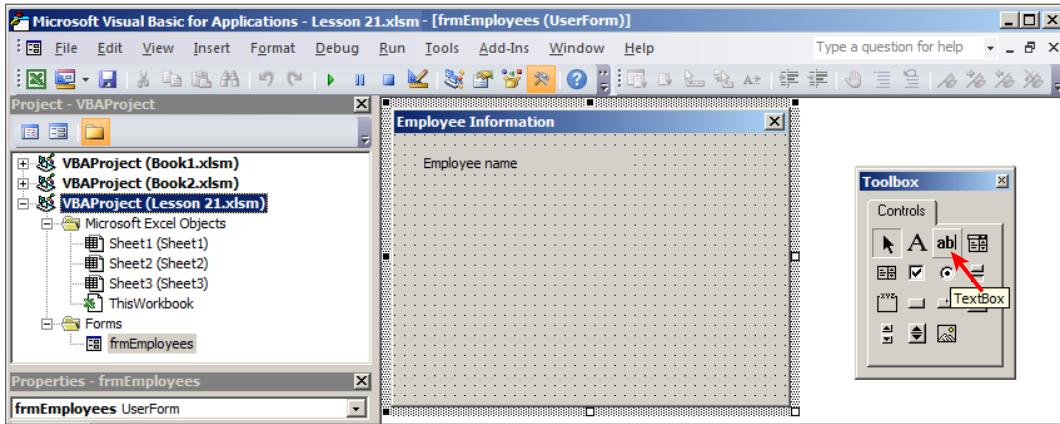


FIGURE 21-10

In Figure 21-13, the Frame's default caption of `Frame1` has been changed to `Position Title`. Now that the Frame's caption is taken care of, Figure 21-13 also shows that the OptionButton icon in the Toolbox is about to be selected. Because an employee would hold only one particular job position title at a time, you can arrange a series of OptionButtons inside the Frame to represent the company's various position titles, where only one can be selected.

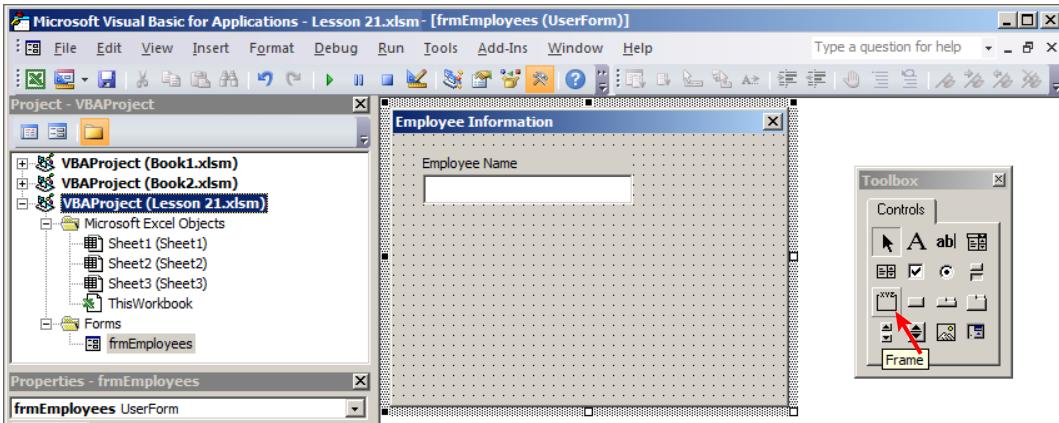


FIGURE 21-11

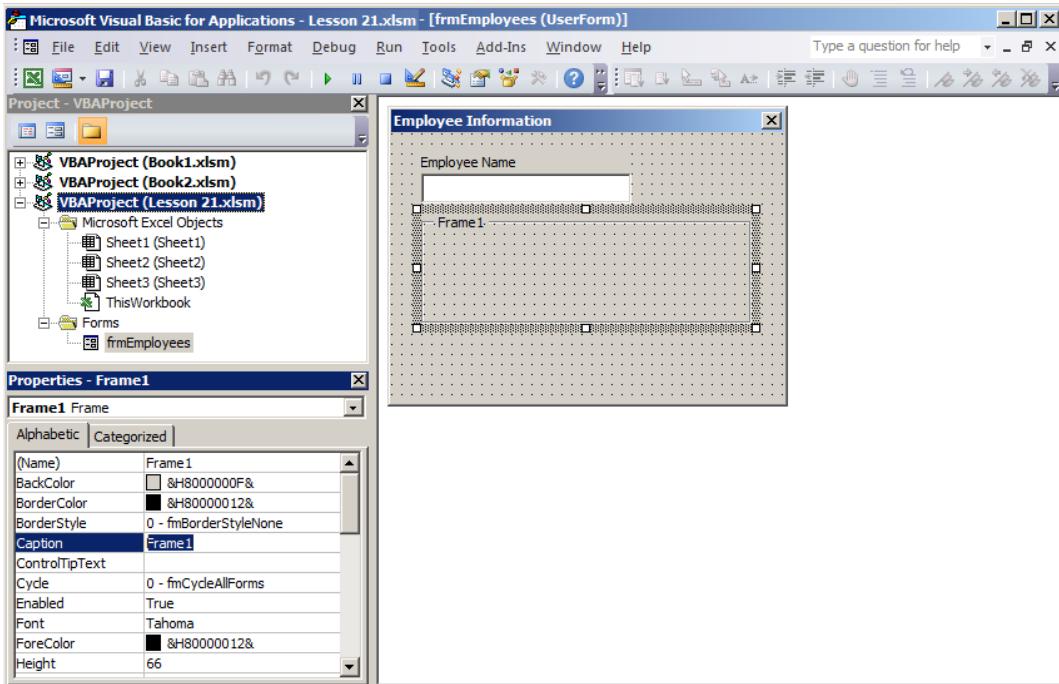


FIGURE 21-12

In this basic UserForm example, Figure 21-14 shows four position titles from which to choose, each as a caption among the four OptionButton controls that were placed inside the Frame. The OptionButtons were added and captioned one at a time. Planning ahead, Figure 21-14 also shows the CommandButton icon in the Toolbox, which is about to be selected so you can add a couple of buttons as the last step in building the UserForm's front-end design.

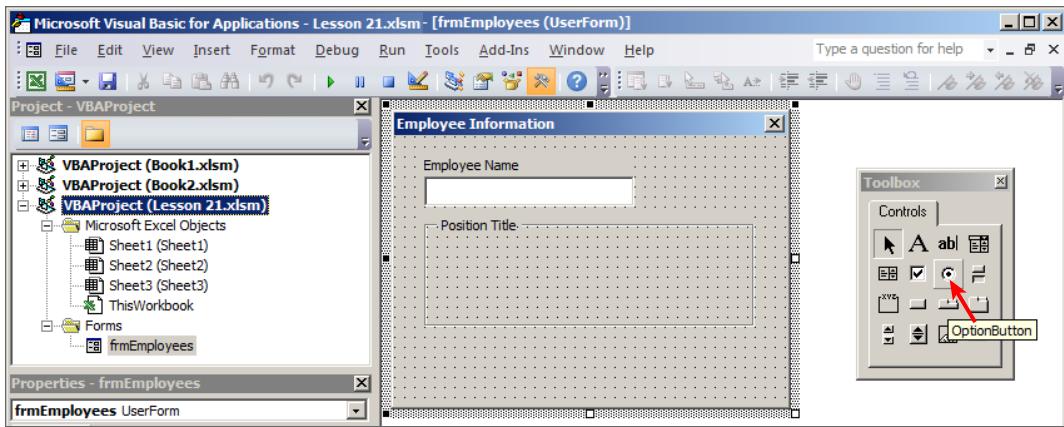


FIGURE 21-13

In Figure 21-15, two CommandButtons have been added, which completes the UserForm's interface design. One of the CommandButtons is captioned OK, which is a common and intuitive caption for users to click their confirmation of data entries. The other CommandButton is a Cancel button to allow users to quit the UserForm altogether, if they so choose.

NOTE A standard of proper UserForm design is to always allow your users an escape route out of the UserForm. This is commonly done with a Cancel or Exit button that users can click when they want to leave the form.

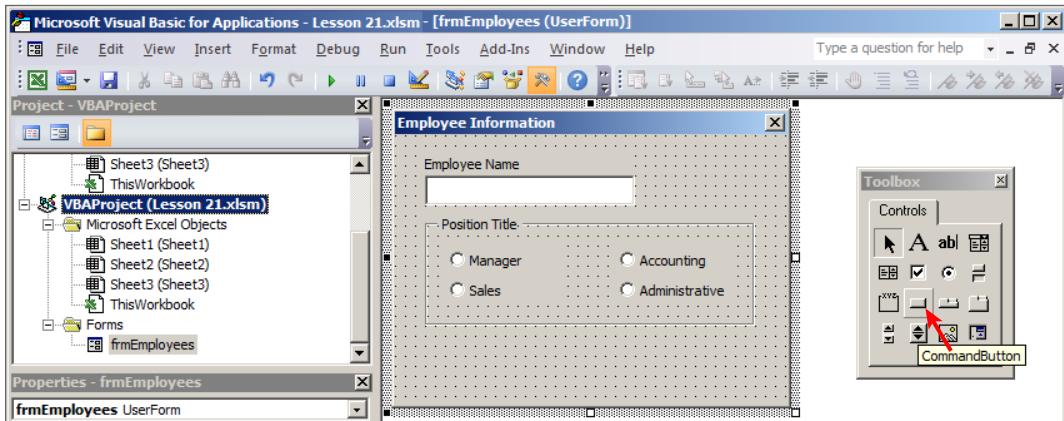


FIGURE 21-14

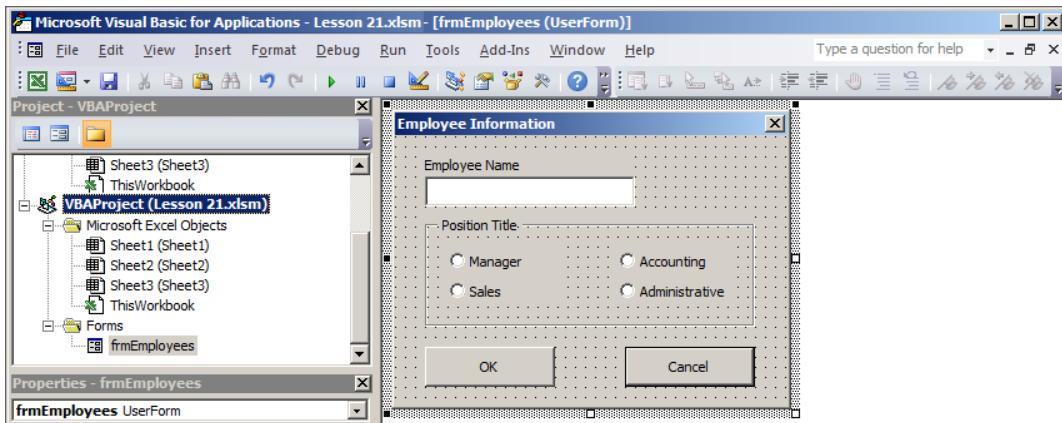


FIGURE 21-15

SHOWING A USERFORM

To show a UserForm, you execute the VBA Show command in a statement with the syntax `UserFormName.Show`. For example, if you had performed the same steps as you've seen in this lesson to create the `frmEmployees` UserForm, you may have a simple macro like this to call the UserForm:

```
Sub EmployeeForm()
    frmEmployees.Show
End Sub
```

If you'd like to see how the UserForm looks when it is called in the actual worksheet environment, without having to write a formal macro for yourself, you can type `frmEmployees.Show` into the Immediate window and press Enter. Figure 21-16 shows how you and your users see the example UserForm.

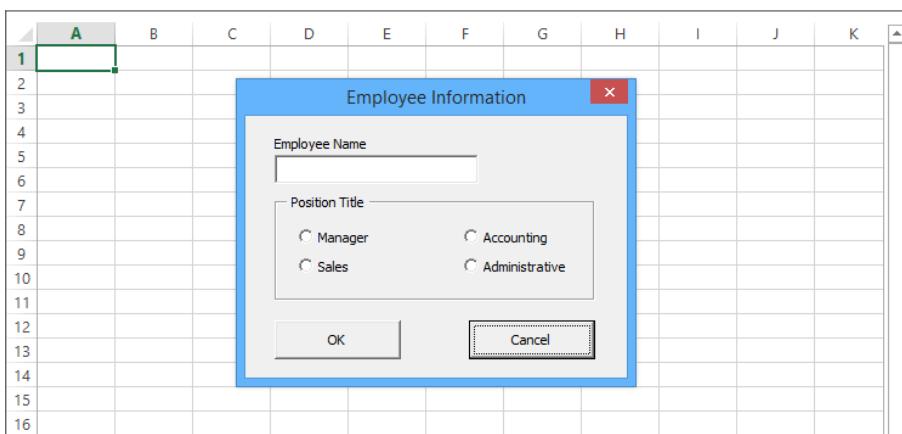


FIGURE 21-16

WHERE DOES THE USERFORM'S CODE GO?

This lesson introduced UserForms and led you through the steps to create a basic form that contains various controls. In Lessons 22 and 23 you see examples of how those and other UserForm controls are programmable with event-driven VBA code.

A UserForm is a class of VBA objects that has its own module. Similar to the notion that each worksheet has its own module, each UserForm you add to your workbook is automatically created with its own module. Accessing a UserForm's module is easy: In the VBE, you can double-click the UserForm itself in the design pane; or in the Project Explorer, you can right-click the UserForm name and select View Code, as shown in Figure 21-17.

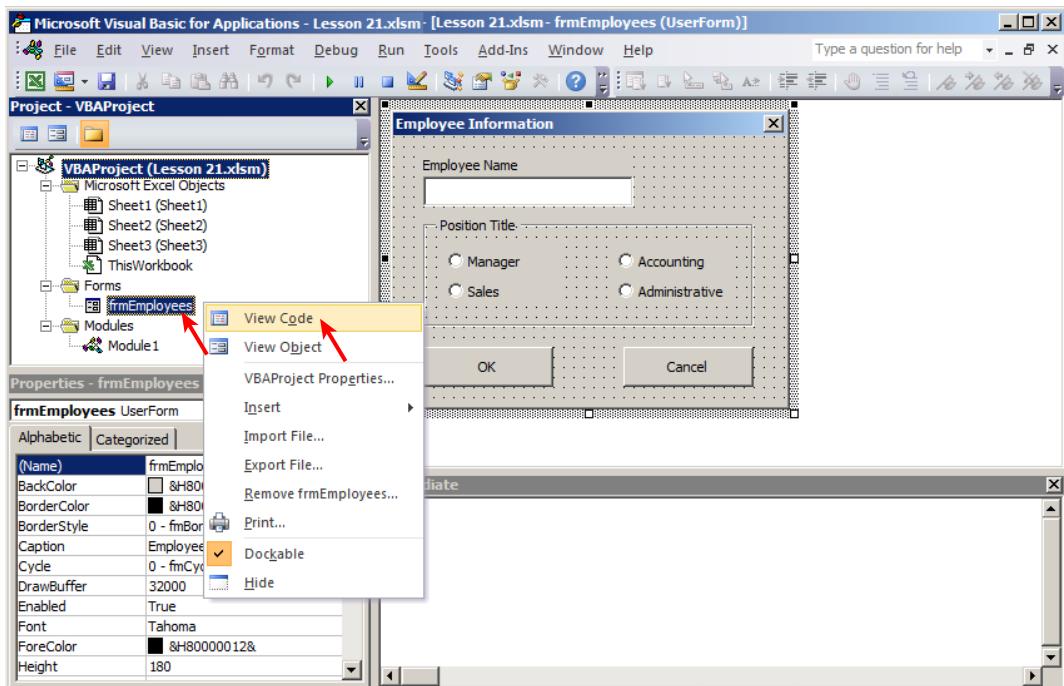


FIGURE 21-17

CLOSING A USERFORM

You have two ways to close a UserForm. One way is with the `Unload` method and the other way is with the `Hide` method. Though both methods make the UserForm look as if it has gone away, they each carry out different instructions. This can be a point of confusion for beginning programmers, so it's important to understand the distinction between `Unload` and `Hide`.

Unloading a UserForm

When you unload a UserForm, the form closes and its entries are cleared from memory. In most cases, that is what you want—for the data that was entered to be recorded in some way, or passed to Public variables, and then closed. The statement that unloads a UserForm is simply `Unload Me`, and it is commonly associated with a CommandButton for that purpose, such as the Cancel button that was placed on this lesson's example UserForm.

Suppose you want to unload the UserForm when the Cancel button is clicked. A quick and easy way to do that is to double-click the CommandButton in the UserForm's design, as shown in Figure 21-18.

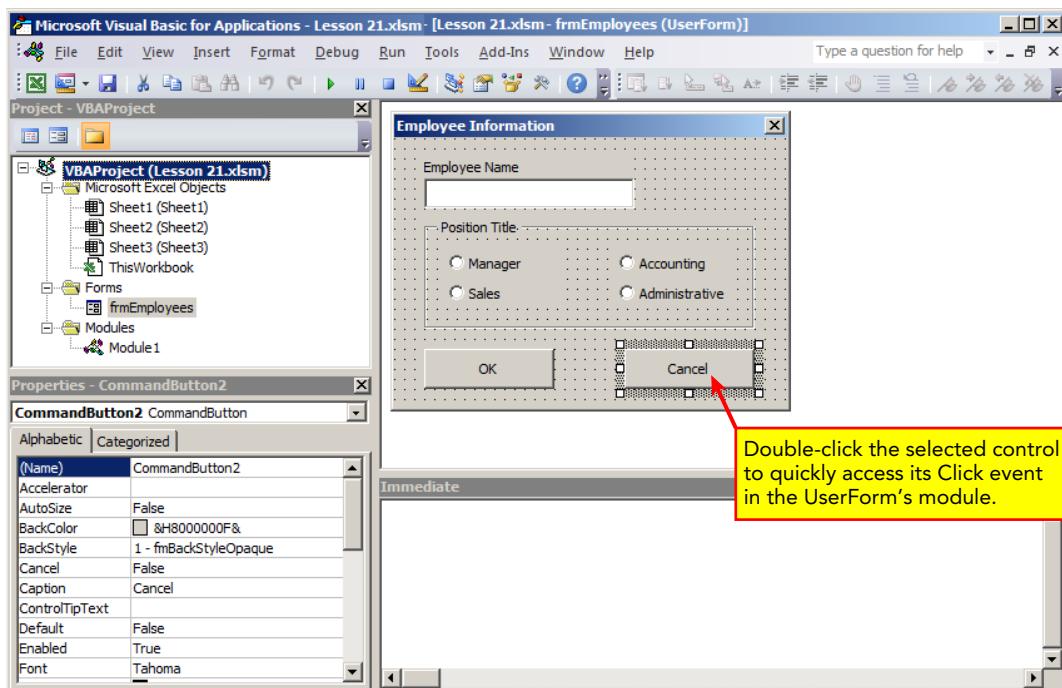


FIGURE 21-18

When you double-click the CommandButton, you see these lines of code in the UserForm's module:

```
Private Sub CommandButton2_Click()
    End Sub
```

To complete the `Click` procedure, type `Unload Me`. When the Cancel button is clicked, the UserForm unloads—that is, it closes and releases from memory the data that was entered—with this `Click` event for that button:

```
Private Sub CommandButton2_Click()
    Unload Me
    End Sub
```

Hiding a UserForm

The `Hide` method makes the UserForm invisible, but the data that was in the UserForm is still there, remaining in memory and able to be viewed when the form is shown again. In some situations you will want this to be the case, such as if you are interacting with two or more UserForms and you want the user to focus on only one form at a time. The statement to hide a UserForm is `Me.Hide`.

NOTE To summarize the difference between `Unload` and `Hide`, the method you choose depends on why you don't want the UserForm to be seen. Most of the time, you'll want the form cleared from memory, but sometimes, information that was entered into the form needs to be referred to the next time you show the form while the workbook has remained open. Closing the workbook automatically unloads a UserForm only if it was hidden.

TRY IT

In this lesson, you design a simple UserForm with a Label control, a TextBox control, a CheckBox control, and two CommandButton controls.

Lesson Requirements

To get the sample workbook, you can download Lesson 21 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Press Alt+F11 to go to the Visual Basic Editor.
2. Select the workbook name in the Project Explorer window, and from the menu bar at the top of the VBE click Insert \Rightarrow UserForm.
3. Select the UserForm in its design window, and press the F4 key (or click View \Rightarrow Properties Window) to show the Properties window.
4. Change the `Name` property to `frmClients` and change the `Caption` property to `Clients`.
5. Size the UserForm by setting its `Height` property to `240` and its `Width` property to `190`.
6. From the menu bar at the top of the VBE, click View \Rightarrow Toolbox.
7. From the Toolbox, click the Label control icon and draw a Label across the top of the UserForm. With the Label control selected, change its `Caption` property to `Company Name`.
8. From the Toolbox, click the TextBox control icon and draw a TextBox directly below the Label.

9. From the Toolbox, click the Label control icon again, and draw a Label a little bit below the TextBox. With that Label control selected, change its `Caption` property to `Client's business -- check all that apply:`.
10. Directly below the Label from Step 9, from the Toolbox, click the CheckBox control icon and draw a CheckBox that is wide enough for you to have its `Caption` property be `Agriculture`.
11. Repeat Step 10 four more times, meaning you'll draw a total of five CheckBoxes that are stacked one above the other in a vertical fashion. Change the `Caption` labels on the four other CheckBoxes to `Manufacturing`, `Medical`, `Retail`, and `Technology`.
12. From the Toolbox, click the CommandButton icon control and draw a CommandButton in the lower-left corner of your UserForm. Change its `Caption` property to `OK`.
13. Draw a second CommandButton in the lower-right corner of your UserForm. Change its `Caption` property to `Cancel`.
14. Take a look at your completed UserForm as it would appear when called. While you are still in the VBE, press `Ctrl+G` to get into the Immediate window. Type `frmClients.Show` and press Enter. Your UserForm should look like the one shown in Figure 21-19. There is no code behind the CommandButtons, so to close this UserForm, click the X Close button at the top-right corner.



FIGURE 21-19

REFERENCE Please select the video for Lesson 21 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

22

UserForm Controls and Their Functions

UserForms enable you to interact with your users in ways that you can't when using standard Message Boxes, InputBoxes, or controls embedded onto your worksheet. With UserForms, you can control the input of information by validating the kind of data that gets entered, the order in which it is entered, and, if your workbook requires it, the exact location where the information should be stored and how it should be recalled. This lesson leads you through the design of various UserForms, with examples of how to program an assortment of controls that you'll utilize most frequently.

UNDERSTANDING THE FREQUENTLY USED USERFORM CONTROLS

As demonstrated in Lesson 21, when you add a UserForm to your workbook, the first thing you see is the empty UserForm in its design window, not unlike a blank canvas upon which you'll strategically place your controls. The controls you utilize depend upon the task at hand, and you'll come across countless sets of circumstances for which a UserForm is the right tool for the job.

Still, you'll find that a core group of frequently used controls can handle most of your UserForm requirements. The fun part is tapping into the events each control supports to create a customizable UserForm that's user-friendly and, most importantly, gets the job done.

NOTE *As you see in Lesson 23, you are not limited to the relatively few controls shown by default on the Toolbox. Dozens more Toolbox controls are available to you, many of which you'll probably never use, but some you eventually will.*

CommandButtons

The CommandButton is a basic staple of just about any UserForm. The combination of a `Caption` property and `Click` event make CommandButtons an efficient way to convey an objective and then carry it out with a mouse click. And if for no other reason, a Cancel or Exit button is about as basic a need as any form will have.

Suppose you want to provide your users with a quick way to print a worksheet in either portrait or landscape orientation. You can make it easy for your users to click a button to indicate their decision, and then just go ahead and execute the print job. Figure 22-1 shows an example of how you can do this, followed by the code behind each of the CommandButtons.

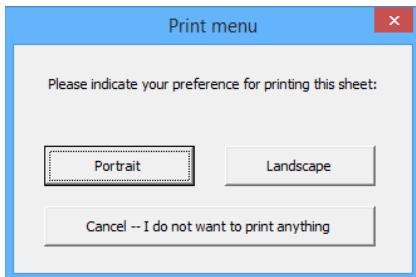


FIGURE 22-1

```
Private Sub cmdPortrait_Click()
With ActiveSheet
    .PageSetup.Orientation = xlPortrait
    .PrintPreview
End With
End Sub

Private Sub cmdLandscape_Click()
With ActiveSheet
    .PageSetup.Orientation = Landscape
    .PrintPreview
End With
End Sub

Private Sub cmdCancel_Click()
Unload Me
End Sub
```

NOTE As you can see in the preceding code, each of the CommandButtons has been named using the prefix `cmd` followed by a notation that gives a clue as to the purpose of the button (see `cmdPortrait_Click()`, `cmdLandscape_Click()`, and `cmdCancel_Click()`). There is nothing sacred about the `cmd` prefix for CommandButtons, or about the `lbl` prefix when naming Labels, or about any naming prefix for that matter. Still, it's wise to name your controls in some intuitive and consistent way so you and others recognize the control and its purpose when reviewing your VBA code.

Labels

You've seen Label controls, such as the examples in Lesson 21, where the Label's `Caption` property is set to always display the same text. Sometimes, a Label can serve to display dynamic information that is not a static piece of text, and in that case, you'd leave the `Caption` property empty.

UserForms have an `Initialize` event that is triggered when you call the UserForm, which can help you take action on your UserForm or workbook. Suppose you want to enhance the customized look of your form with a welcome greeting that changes to reflect the time of day. For example, if the UserForm were to be opened in the morning, the message would include the text `Good morning`, and so on for the afternoon and evening. The following code achieves the effect shown in Figure 22-2:

```
Private Sub UserForm_Initialize()
Dim TymeOfDay As String
If Time < 0.5 Then
    TymeOfDay = "Good Morning ! "
ElseIf Time >= 0.5 And Time < 0.75 Then
    TymeOfDay = "Good Afternoon ! "
Else
    TymeOfDay = "Good Evening ! "
End If
Label1.Caption = TymeOfDay & "Welcome to the company workbook."
End Sub
```

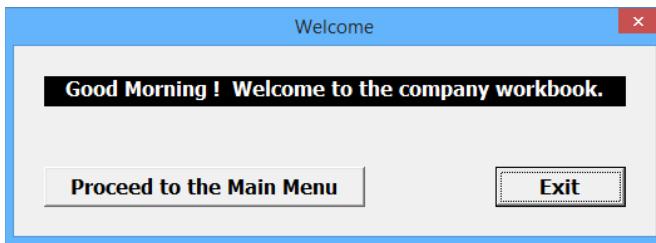


FIGURE 22-2

TIMES IN VBA

Even after studying the preceding code, you might wonder why a number less than .5 translates to morning, why a number greater than or equal to .5 and less than .75 translates to afternoon, and why a number greater than or equal to .75 translates to evening. The reason is that VBA regards a time of day as a completed percentage of the calendar day. For example, 12:00 noon is the halfway mark of a calendar day, and one-half of something can be mathematically represented by the expression `.5`. The `Time` function in VBA interprets a number less than `.5` as morning because by definition, half the day would not yet have completed. Afternoon is between `.5` (12:00 noon) and up to just before 6:00 PM, which the `Time` function interprets as `.75`, being at the three-fourths mark of the 24-hour calendar day. A `Time` number greater than or equal to `.75` is evening because it is at or past 6:00 PM and before the `Time` number of `0`, which is 12:00 midnight of the next day.

You can also populate a Label's caption from another control's event procedure. Suppose your UserForm provides a CommandButton that when clicked, toggles column C as being visible or hidden, such as with this line of code in the CommandButton's Click event:

```
Columns(3).Hidden = Not Columns(3).Hidden
```

NOTE *Columns(3) is another way of expressing Columns("C:C"). The 3 refers to C being the third letter in the alphabet, which corresponds to the third column from the left in the worksheet grid. If it were column D, the syntax notation would be Columns(4) and so on. There is no schematic advantage to using one style of expression over the other, but I included the numeric expression here so you can be aware of it, and use it in your macros if it feels more intuitive for you to do so.*

It's a good practice when constructing UserForms to give the users an indication that confirms what they've just done. In this example, a Label control can be near the CommandButton that confirms the visible or hidden status of column C, with the following code:

```
Private Sub CommandButton1_Click()
Columns(3).Hidden = Not Columns(3).Hidden
Label1.Caption = "Column C is " &
    IIf(Columns(3).Hidden = True, "hidden", "visible")
End Sub
```

TextBoxes

A TextBox is most commonly used to display information that is entered by a user, or is associated with a cell through the TextBox's ControlSource property, or is entered programmatically, such as to display a calculation result or a piece of data from a worksheet table. You have probably seen TextBoxes when you've entered information on electronic forms, such as when you've entered your name, address, and credit card number when making a purchase online.

Figure 22-3 shows a UserForm with three TextBox controls. In this example, I've entered my first and last name, and a password that is represented in the figure as a series of asterisks. UserForms are a good way to greet your user and ask for a password with a TextBox, and with the TextBox's PasswordChar property, you can set any character (in this case an asterisk) to appear instead of the password, so no one else sees the password as it is being typed.

NOTE *Formatting of TextBoxes is limited to the entire TextBox entry. For example, if you want any portion of the TextBox's contents to be bold, the entire contents must be bold.*

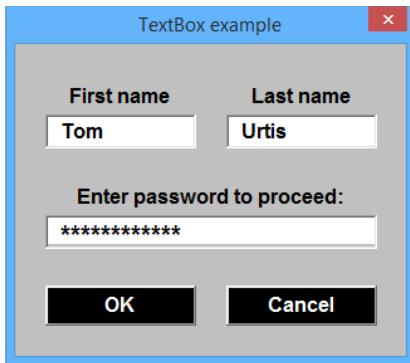


FIGURE 22-3

Sometimes you will want a TextBox to accept only numeric entries, such as a dollar figure, a calendar year, or a person's age in years. The following code monitors each keystroke entry into TextBox1, and disallows any character that is not a number. As a courtesy to the user, a message appears to immediately inform the user that an improper character was attempted and disallowed:

```
Private Sub TextBox1_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
Select Case KeyAscii
Case 48 To 57
Case Else
KeyAscii = 0
MsgBox "You typed a non-numeric character", _
vbExclamation, _
"Numbers only, please!"
End Select
End Sub
```

NOTE In the preceding code example, you might not be familiar with the term "ASCII" (pronounced "askee"), which is an acronym for American Standard Code for Information Interchange. Computers can only understand numbers, so a numerical representation is needed for alphanumeric characters and other symbols such as # and @. In the preceding code, numbers 0–9 are recognized by virtue of their ASCII representation of 48–57. If you'd like to see a list of all 255 ASCII and Extended ASCII characters, you can produce it yourself on an Excel worksheet by entering the formula =CHAR(ROW()) in cell A1, and copying it down to cell A255. Each cell holds a character (some characters will not be visible) whose ASCII number corresponds to the cell's row number.

TextBoxes can display calculated results, and when using numbers for mathematical operations, you need to use the Val function, which returns the numbers contained in a TextBox string as a numeric value. Suppose your UserForm contains seven TextBoxes into which you enter the sales dollars for

each day of the week. As shown in Figure 22-4, an eighth TextBox can display the sum of those seven numbers when a CommandButton is clicked, with the following code:

```
Private Sub CommandButton1_Click()
Dim intTextBox As Integer, dblSum As Double
dblSum = 0
For intTextBox = 1 To 7
dblSum = dblSum + Val(Controls("TextBox" & intTextBox).Value)
Next intTextBox
TextBox8.Value = Format(dblSum, "#,###")
End Sub
```

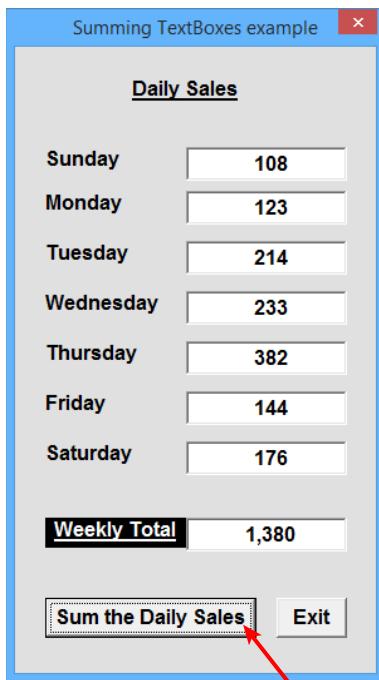


FIGURE 22-4

ListBoxes

A ListBox displays a list of items and lets you select one or more. ListBoxes are fairly versatile in their display of information and their options for allowing you to select one, many, or all listed items.

Suppose you want to list all 12 months of the year, so any particular month can be selected to perhaps run a report for income and expenses during that month. You might also want the flexibility to run a single report that includes activity for any combinations of months. The ListBox control is an excellent choice because you can set its `MultiSelect` property to allow just one item, or multiple items, to be selected. Figure 22-5 shows an example of how you can control the way the items appear with the `ListStyle` property, and selection options for your ListBox (allow only one or more than one item to be selected) with the `MultiSelect` property.

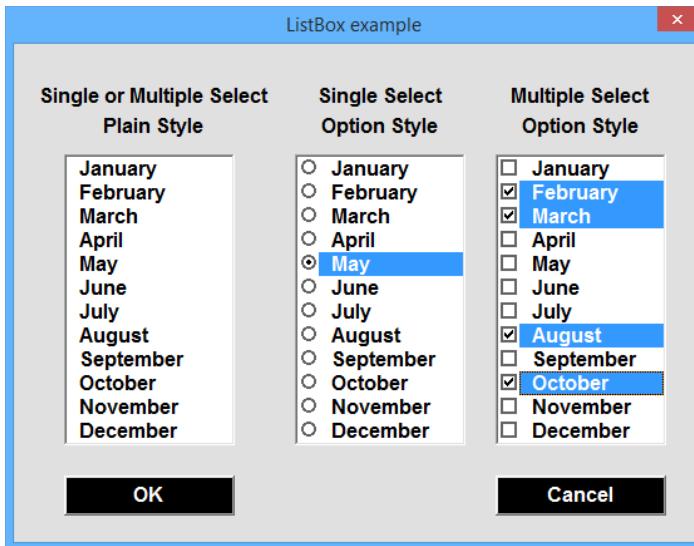


FIGURE 22-5

You can use two common methods to populate a ListBox with items. In the preceding example, the 12 months of the year could be listed on a worksheet, say on Sheet2 in range A1:A12. To have the ListBox display the list of months, you can enter `Sheet2!A1:A12` as the RowSource property for that ListBox.

In many cases, however, you'll want to populate your ListBox without having to store the items on a worksheet. The UserForm's Initialize event is perfect for populating your ListBox with a dynamic or static list of items. Suppose you want to list the names of various countries. The following code does that using the AddItem method in the UserForm's Initialize event, which you can easily append when you want to add or omit a country name:

```
Private Sub UserForm_Initialize()
    With ListBox1
        .RowSource = ""
        .AddItem "England"
        .AddItem "Spain"
        .AddItem "France"
        .AddItem "Japan"
        .AddItem "Australia"
        .AddItem "United States"
    End With
End Sub
```

NOTE When you populate a ListBox programmatically (or, as you see later, a ComboBox), be sure to clear the control's RowSource property or you will get a runtime error when you call (initialize) the UserForm. This was done in the preceding code by setting RowSource equal to an empty string.

The following code lists all the visible worksheets in your workbook, and excludes the worksheets that are hidden:

```
Private Sub UserForm_Initialize()
With ListBox1
.Clear
Dim wks As Worksheet
For Each wks In Worksheets
If wks.Visible = xlSheetVisible Then .AddItem wks.Name
Next wks
End With
End Sub
```

ListBoxes support many events, and using the `Click` event, for example, this code activates the worksheet whose name you click, with the `ListBox`'s `MultiSelect` property set to `0-fmMultiSelectSingle`:

```
Private Sub ListBox1_Click()
Worksheets(ListBox1.Value).Activate
End Sub
```

ComboBoxes

A ComboBox combines the features of a ListBox and a TextBox, in that you can select an item from its drop-down list, or you can type an item into the ComboBox that is not included in its list. Most of the time, you'll use the ComboBox the same way you'd use data validation, where a drop-down arrow is visible for revealing the list of items that are available for selection.

NOTE If you want to limit the ComboBox to only accept items from the drop-down list, set its `Style` property to 2 - `fmStyleDropDownList`.

ComboBoxes allow only one item to be selected; you cannot select multiple items in a ComboBox the way you can with a ListBox. However, ComboBoxes are populated much the same way as ListBoxes, with a `RowSource` property and an `AddItem` method.

Suppose you want to guide the users of your workbook to select a year that is within three years—past or future—of the current year. The following code could accomplish that, with Figure 22-6 showing the ComboBox's list after the drop-down arrow was clicked, assuming the current year is 2015:

```
Private Sub UserForm_Initialize()
With ComboBox1
.Clear
Dim iYear As Integer, jYear As Integer
jYear = Format(Date, "YYYY")
For iYear = 1 To 7
ComboBox1.AddItem jYear - 3
Next iYear
End Sub
```

```
jYear = jYear + 1
Next iYear
End With
End Sub
```

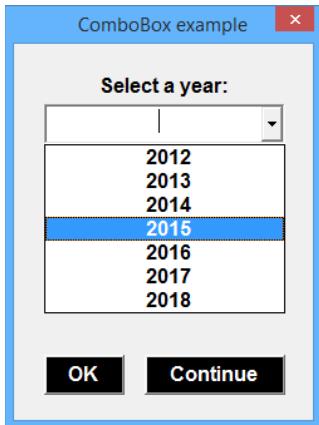


FIGURE 22-6

As with a ListBox, if the items needed to populate the ComboBox are listed on a worksheet, it does not mean you must refer to them with the RowSource property. You can leave the RowSource property empty, and populate the ComboBox (same concept applies to a ListBox) with the following code example, assuming the values are listed in range A1:A8 with no blank cells in that range:

```
Private Sub UserForm_Initialize()
ComboBox1.List = Range("A1:A8").Value
End Sub
```

NOTE If you want the first item in the drop-down list to be automatically visible in your ComboBox, you can add the following line before the End Sub line, assuming the ComboBox is named ComboBox1:

```
ComboBox1.ListIndex = 0
```

Sometimes you need to populate the ComboBox (or ListBox) with items listed in a range that also contains blank cells. Figure 22-7 shows how horrible that makes the drop-down list look if you attempted to populate the ComboBox with the line of code `ComboBox1.List = Range("A1:A8").Value`.

Much nicer looking is Figure 22-8, which does not show empty spaces in its drop-down list even though empty cells exist among the list of names. The code to do that is shown here, which uses the LEN function to disregard cells that have no value in them:

```
Private Sub UserForm_Initialize()
Dim LastRow As Long, cboCell As Range
LastRow = Cells(Rows.Count, 1).End(xlUp).Row
For Each cboCell In Range("A1:A" & LastRow)
If Len(cboCell) > 0 Then ComboBox1.AddItem cboCell.Value
Next cboCell
End Sub
```

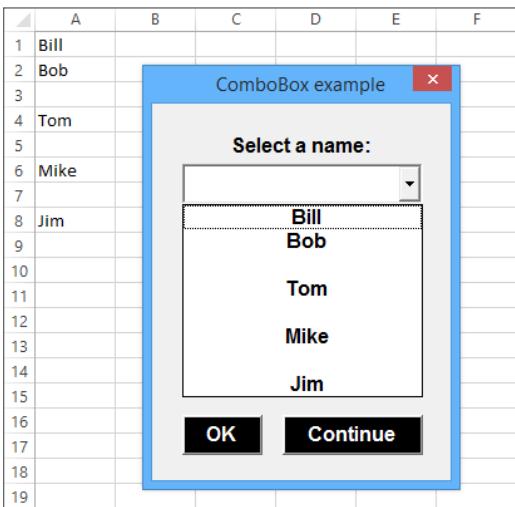


FIGURE 22-7

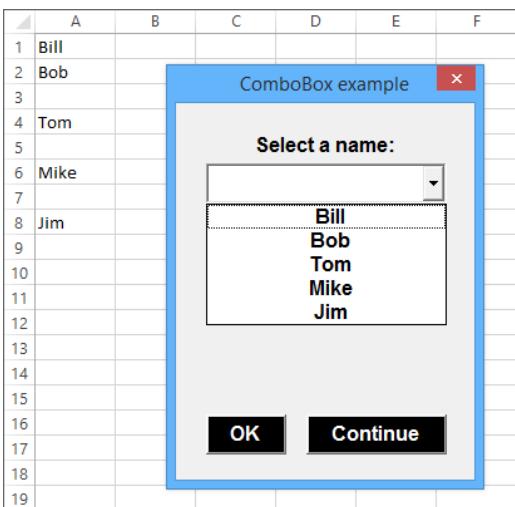


FIGURE 22-8

Check Boxes

A CheckBox on your UserForm can serve one of two purposes: to provide users with an option that is of the Yes/No variety, without a superfluous message box to present the option, or to provide a pair of OptionButtons (covered in the next section). Simply, a single CheckBox is inferred to mean Yes or OK if it is checked, and No if it is not checked.

As you develop more complex UserForms, you will want to provide your users with convenient options for viewing—or not viewing—interface objects that might be useful to them in some cases, and irrelevant in others. For example, Figure 22-9 shows the same UserForm in two situations, where the user can check or uncheck the CheckBox captioned Show List of Months. If the CheckBox is unchecked, neither the ListBox nor the Label above it will be visible, but if the CheckBox is checked, those controls do appear. The code associated with the CheckBox follows:

```
Private Sub CheckBox1_Click()
With CheckBox1

If .Value = True Then
    Label1.Visible = True
    ListBox1.Visible = True

Else

    Label1.Visible = False
    ListBox1.Visible = False

End If

End With
End Sub
```

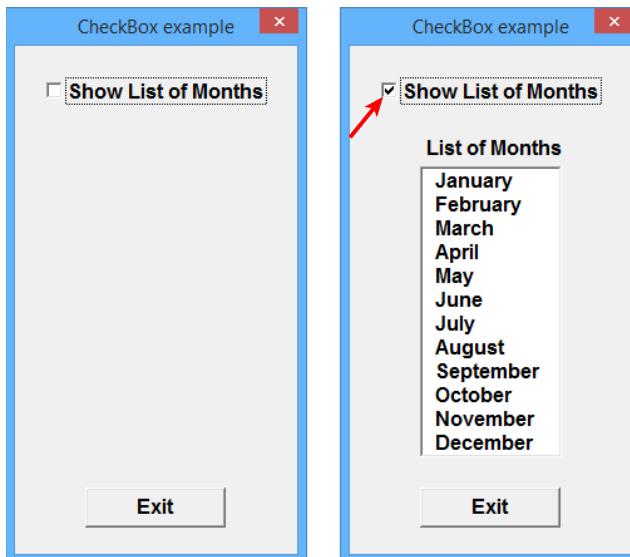


FIGURE 22-9

NOTE Users appreciate having a say as to what they see on a form, which helps give them some control over the form's navigation process. However, as the workbook's developer, your primary objective is to design a smart form. In this example, if the selection of a month name is a mandatory action in the UserForm's overall process, you would not consider building in the option of hiding a ListBox of month names. You'll often see a single CheckBox on a UserForm when a simple preference is to be indicated, such as including a header on all printed pages, or performing the same action on all worksheets.

Another popular use of CheckBoxes is to provide the user with several options at the same time. Figure 22-10 shows a UserForm that asks for users to indicate which regions a company report should include. When the OK button is clicked, you can assign variables to each CheckBox that was checked, and incorporate those variables later in a VBA decision process that recognizes only the checked regions. One way to accomplish that is to loop through each CheckBox and identify the selected CheckBox(es), as shown in the following code:

```
Private Sub cmdOK_Click()
    'Declare an Integer type variable for the five Checkboxes.
    Dim intCheckBox As Integer
    'Declare a String type variable for the list of selected Checkboxes.
    Dim strCheckBoxNames As String
    'Open a For next loop to examine each of the 5 Checkboxes.
    For intCheckBox = 1 To 5
        'If the CheckBox is selected, meaning its value is True,
        'build the strCheckBoxNames string with the caption of the
        'selected CheckBox, followed by a Chr(10) new line character
        'for readability in the confirming MsgBox.
        If Controls("CheckBox" & intCheckBox).Value = True Then
            strCheckBoxNames = strCheckBoxNames & _
                Controls("CheckBox" & intCheckBox).Caption & Chr(10)
        End If
        'Continue the loop until all 5 Checkboxes have been examined.
    Next intCheckBox
    'Display a Message Box to advise the users what they selected.
    MsgBox strCheckBoxNames, , "Regions that were checked:"
End Sub
```

OptionButtons

An OptionButton is used when you want the user to select one choice from a group of optional choices. You would use a group of OptionButtons to show the single item that was selected among the group's set of choices. For example, on a college application form, in the gender section, an applicant could select only Male or Female.

In Figure 22-11, a menu for running a financial report might ask the user to select the month of activity upon which the report should be based. A group of 12 OptionButtons limits the user to only one selection. Each OptionButton's Caption property was filled in with the name of a month.

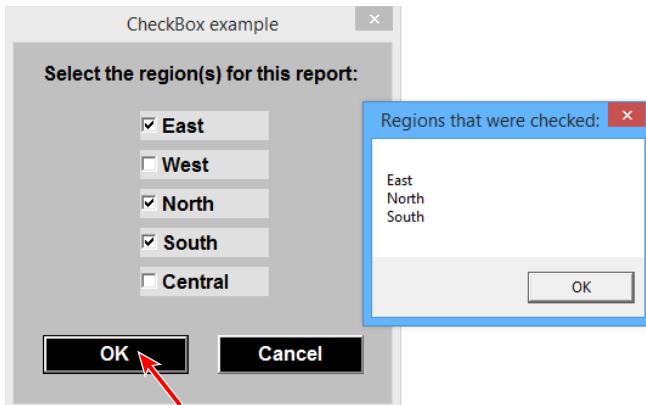


FIGURE 22-10

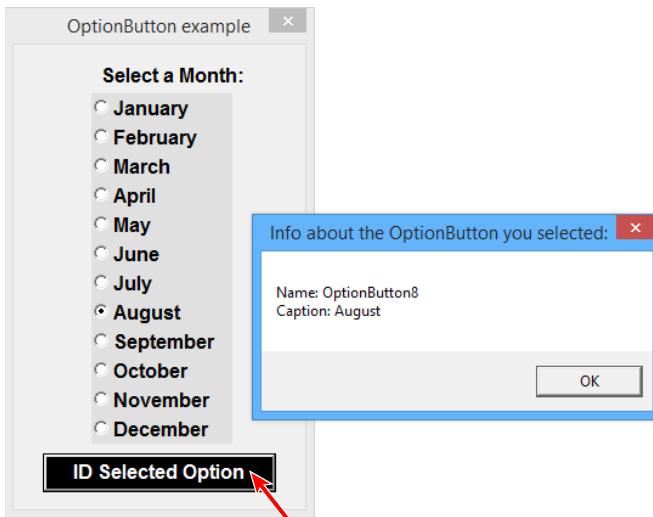


FIGURE 22-11

Figure 22-11 shows that the month of August was selected, and in real practice, you'd identify that selection in your code with a variable that refers to the selected month name, and produces the report for that month. One way to do that is to loop through each of the OptionButtons and stop when you encounter the selected OptionButton whose value would be True.

To help make the point, there is a button on the form with the caption ID Selected Option, and when you click the button, a message box appears, telling you the name of the selected OptionButton and its caption. The following code examines the status of the OptionButtons and then produces the message box:

```
Private Sub CommandButton1_Click()
Dim intOption As Integer, optName As String, optCaption As String
For intOption = 1 To 12
```

```
If Controls("OptionButton" & intOption) = True Then
    optName = Controls("OptionButton" & intOption).Name
    optCaption = Controls("OptionButton" & intOption).Caption
    MsgBox _
        "Name: " & optName & vbCrLf & _
        "Caption: " & optCaption, , -
        "Info about the OptionButton you selected."
    Exit For
End If
Next intOption
End Sub
```

OptionButtons have a useful property called `GroupName` that you should be aware of. In Figure 22-11, a simple UserForm lists 12 OptionButtons, all with the same objective of eliciting a selection for a particular month. But what if your UserForm has other sections for user options that require OptionButtons, such as to select a day of the week, or a print orientation preference of Landscape or Portrait? You'll find many reasons to apply OptionButtons to your UserForms, and you need each set of options to be a mutually exclusive group.

You have two ways to create a group of mutually exclusive OptionButton controls. You can place the group inside a Frame (a control that is covered in the next section), or you can use the `GroupName` property of the related OptionButtons to group them together. In Figure 22-12, the OptionButtons have been selected in the UserForm's design window, and the `GroupName` property has been defined with the name `Months`.

NOTE *Whether organized by GroupName or a Frame control, clicking an OptionButton sets its value to True and automatically sets the other OptionButtons in the group (or in the Frame) to False.*

Frames

Frame controls group related controls together to provide an organized look and feel when the UserForm calls for many controls. Figure 22-13 illustrates an example of employing a Frame.

When you place controls within a Frame control, manipulating the Frame's properties can affect all the controls inside the Frame. For example, assuming the Frame control shown in Figure 22-13 is named `Frame1`, this line of code would hide that frame along with all the controls inside it:

```
Frame1.Visible= False
```

Sometimes you want your Frame to be visible, but you want all the controls inside the Frame to be temporarily disabled. You can disable the Frame and render its controls unusable with the following line of code:

```
Frame1.Enabled = False
```

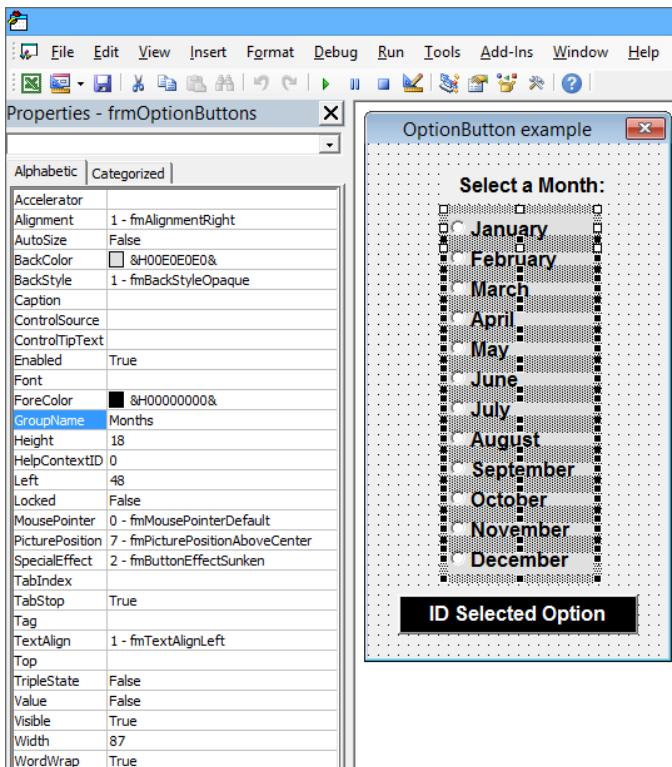


FIGURE 22-12

If you test that, you see a curious result, which is the controls inside the Frame are not “grayed out” but are essentially disabled because they are rendered useless by virtue of the Frame being disabled. The controls themselves appear to be enabled, which can fool your users into wondering what’s wrong with perfectly normal-looking controls that do not respond to any keystrokes or mouse clicks.

If you want to disable the actual controls inside the Frame and make them look disabled, you must loop through each of the controls inside the Frame with the following example code. Note that this code does not disable Frame1, only the controls inside it:

```
Dim FrmControl As Control
For Each FrmControl In Frame1.Controls
    FrmControl.Enabled = False
Next FrmControl
```

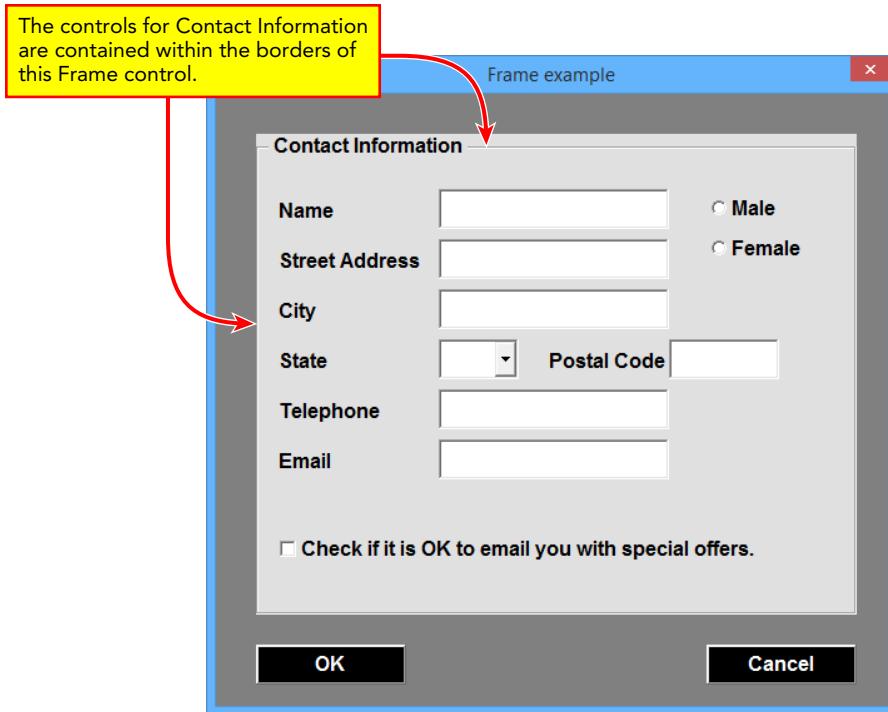


FIGURE 22-13

Naturally, to enable a control that's been disabled, change the `False` statement to `True`, which you can handle in a separate procedure, or in one single procedure with a line of code that toggles the `Enabled` property using the `Not` statement. The following example shows how to do this:

```
Private Sub CommandButton4_Click()
Dim FrmControl As Control
For Each FrmControl In Frame1.Controls
FrmControl.Enabled = Not FrmControl.Enabled
Next FrmControl
End Sub
```

MultiPages

A MultiPage control is like having a set of tabbed folders that each contain information and controls that would be too voluminous to fit comfortably within the UserForm's interface. Figure 22-14 shows an example of how a MultiPage control can come in handy when a lot of information is being sought from the workbook's users about their viewing preferences.

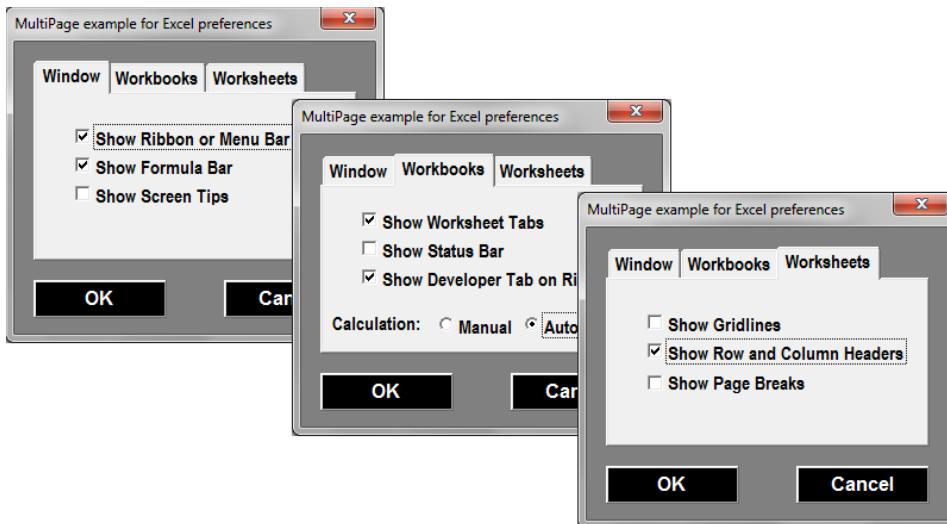


FIGURE 22-14

The MultiPage control has a collection of `Page` objects that are each dedicated to a theme. You can right-click a tab to add a new page, delete the page you right-clicked, rename the page's caption, or move the page. MultiPage controls are a terrific way to maximize the space on your UserForm with a smart, organized look and feel.

TRY IT

In this lesson, you design a UserForm with several controls, including a ListBox that is populated dynamically with the ability to select multiple items.

Lesson Requirements

To get the sample workbook, you can download Lesson 22 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open a new workbook and activate Sheet1.
2. In column A, enter the items in the cells as you see them displayed in Figure 22-15.

	A	B
1	Milk	
2	Eggs	
3		
4	Fruit	
5		
6	Steak	
7		
8	Vegetables	
9	Bread	
10	Oatmeal	
11		
12	Potato Chips	
13	Beer	
14		
15	Cheese	

FIGURE 22-15

3. Press Alt+F11 to get into the Visual Basic Editor.
4. Select your workbook name in the Project Explorer, and from the menu bar click Insert \Rightarrow UserForm and accept its default name of UserForm1.
5. Change the UserForm's Caption property to **Shopping List**.
6. Select the UserForm in its design window, and if the Toolbox is not visible, click View \Rightarrow Toolbox.
7. Draw a ListBox on the UserForm and accept its default name of ListBox1. Set its MultiSelect property to 1 - **fmMultiSelectMulti**.
8. Draw a CommandButton on the UserForm below the ListBox and accept its default name of CommandButton1. Change its Caption property to **Transfer selected items to Sheet2 column E**.
9. Draw another CommandButton on the UserForm below the first CommandButton, and change its Caption property to **Exit**. That completes the design of the UserForm, which should resemble Figure 22-16 when it is called.
10. Double-click the UserForm to go to its module. Type the code under the UserForm's Initialize event that populates the ListBox with items in column A of Sheet1, ignoring the empty cells:

```
Private Sub UserForm_Initialize()
Dim LastRow As Long, ShoppingListCell As Range
With Worksheets("Sheet1")
LastRow = .Cells(Rows.Count, 1).End(xlUp).Row
For Each ShoppingListCell In .Range("A1:A" & LastRow)
If Len(ShoppingListCell) > 0 Then ListBox1.AddItem ShoppingList
Cell.Value
Next ShoppingListCell
End With
End Sub
```

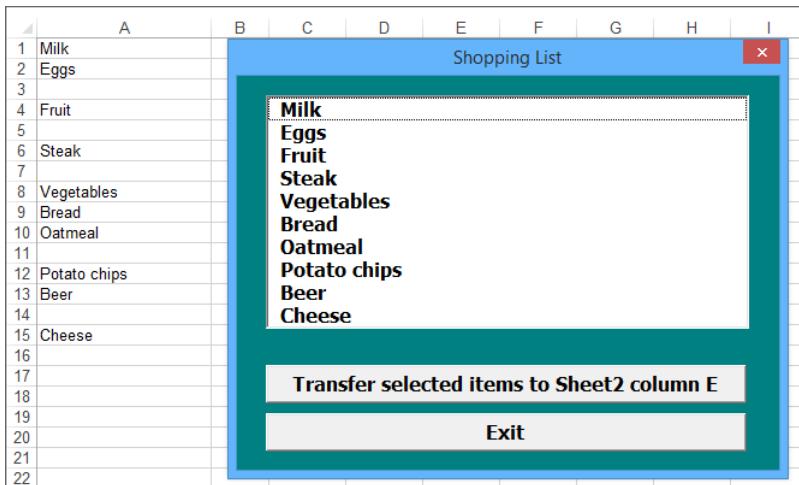


FIGURE 22-16

11. While in the UserForm's module, type the code for CommandButton2 that is the Exit button:

```
Private Sub CommandButton2_Click()
Unload Me
End Sub
```

12. Immediately above the Code window are two drop-down lists. Click the drop-down arrow at the left belonging to the Object field, and select CommandButton1 to place these two statements in the UserForm's module:

```
Private Sub CommandButton1_Click()
End Sub
```

13. For the first line of code in the CommandButton1 Click event, open a With structure for Sheet2, which is the destination sheet for selected items:

```
With Worksheets("Sheet2")
```

14. Declare variables for the ListBox's items and NextRow:

```
Dim intItem As Integer, NextRow As Long
```

15. Clear column E of Sheet2 to start your shopping list with a clean slate:

```
.Columns(5).Clear
```

16. Put a header in cell E1 of Sheet2, to start the list:

```
.Range("E1").Value = "Shopping List"
```

17. Define the NextRow variable as 2, because column E was just cleared and the Shopping List header is in cell E1 with nothing below it:

```
NextRow = 2
```

18. Loop through all items in `ListBox1` and if any are selected, list them in turn in column E of Sheet2:

```
For intItem = 0 To ListBox1.ListCount - 1
If ListBox1.Selected(intItem) = True Then
    .Range("E" & NextRow).Value = ListBox1.List(intItem)
```

19. Add 1 to the `NextRow` variable to prepare for the next selected item:

```
NextRow = NextRow + 1
End If
```

20. Continue the loop until all `ListBox` items have been examined:

```
Next intItem
```

21. Close the `With` structure for Sheet2:

```
End With
```

22. Your final `CommandButton1` code looks like this:

```
Private Sub CommandButton1_Click()
'Open a With structure for Sheet2
With Worksheets("Sheet2")

    'Declare variables for ListBox items and NextRow
    Dim intItem As Integer, NextRow As Long
    'Clear column E of Sheet2
    .Columns(5).Clear
    'Put a header in cell E1
    .Range("E1").Value = "Shopping List"
    'Define the NextRow variable as 2
    'because column E was just cleared and the Shopping List
    'header is in cell E1 with nothing below it.
    NextRow = 2

    'Loop through all items in ListBox 1 and if any are selected,
    'list them in turn in column E of Sheet2.
    For intItem = 0 To ListBox1.ListCount - 1
        If ListBox1.Selected(intItem) = True Then
            .Range("E" & NextRow).Value = ListBox1.List(intItem)
        'Add 1 to the NextRow variable to prepare for the next selected item.
        NextRow = NextRow + 1
    End If
    'Continue the loop until all ListBox items have been examined.
    Next intItem
    'Close the With structure for Sheet2.
End With
End Sub
```

REFERENCE Please select the video for Lesson 22 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

23

Advanced UserForms

Lesson 21 introduces UserForms and shows you how to add controls to your form. Lesson 22 provides several examples of UserForms with frequently used controls to help you gather and store information. This lesson takes an expanded look at how you can get more out of UserForms by tapping into their capacity for supporting some interesting and useful operations.

THE USERFORM TOOLBAR

The Visual Basic Editor has a handy toolbar for working with UserForms, aptly named the UserForm toolbar, shown in Figure 23-1. To display it in the VBE, from the menu bar click View \Rightarrow Toolbars \Rightarrow UserForm.

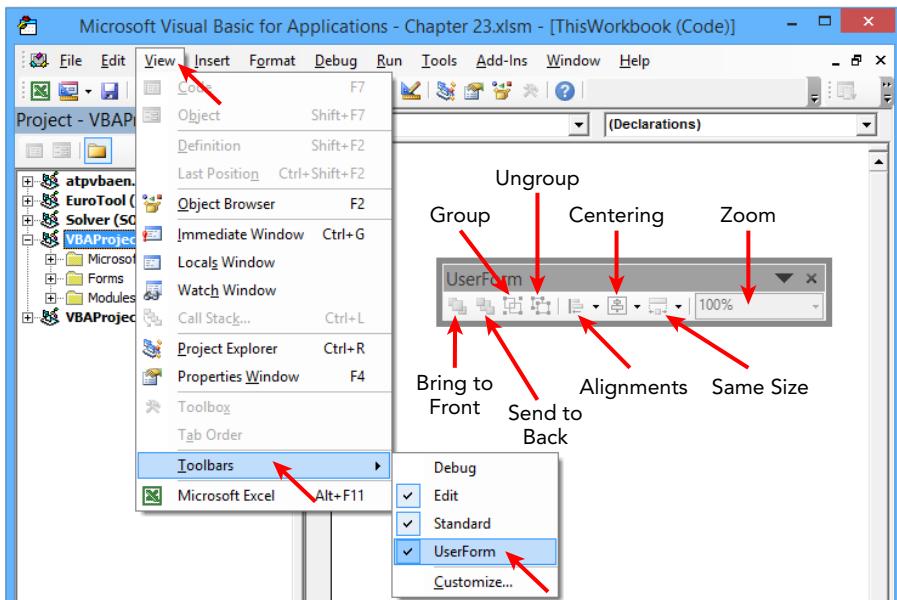


FIGURE 23-1

The UserForm toolbar has eight buttons:

- **Bring to Front:** Brings the selected control to the front of the other controls.
- **Send to Back:** Sends the selected control to the back of the other controls.
- **Group:** Groups the selected controls.
- **Ungroup:** Ungroups the selected grouped controls.
- **Alignments:** The small drop-down arrow to the right of the Alignments icon provides options for aligning the selected controls by their Rights, Lefts, Centers, Tops, Middles, Bottoms, and To Grid.
- **Centering:** Centers the selected controls horizontally or vertically on the UserForm.
- **Same Size:** Sizes the selected controls to be of the same Height, Width, or Both.
- **Zoom:** Displays the UserForm as a zoomed percentage of its normal size.

NOTE If you're working in a UserForm module and you forget the names of controls and you've selected the Require Variable Declaration option (on the Editor tab when you click Tools ➔ Options in the VBE), type Me followed by a dot. You see a list of all the methods and properties for the UserForm, including the list of control names belonging to the UserForm.

MODAL VERSUS MODELESS

Beginning with Excel version 2000, UserForms became equipped with a new property called `ShowModal`. When a UserForm's `ShowModal` property is set to True—that is, when it is shown as Modal—it means that while the UserForm is visible, you cannot select a worksheet cell, another worksheet tab, or any of the Ribbon or menu icons until you close the UserForm. Most of the time, this is what you want—for the UserForm to command all focus and attention while it is visible.

At times the users of your project will benefit from the ability to select cells and generally to navigate worksheets while a UserForm is visible. When that's what you need, call the UserForm by specifying the `ShowModal` property as False. For example:

```
Sub ShowUserForm1()
    UserForm1.Show vbModeless
End Sub
```

You can also write the preceding code line as `UserForm1.Show 0`. The default setting for the `ShowModal` property is `vbModal` (or the numeral 1), which you don't need to specify when calling a UserForm if you want it to be modal. The code line `UserForm1.Show vbModal`, `UserForm1.Show 1`, or (which you have typically been using all along) `UserForm1.Show` shows the UserForm as modal.

NOTE Here's a neat trick that might interest you. When you call a UserForm as modeless, the UserForm is the active object and an extra mouse click is required to actually activate the worksheet. If you want the worksheet itself to be the active object without manual intervention, add the line AppActivate ("Microsoft Excel") below the Show line; here is a full macro example:

```
Sub ShowUserForm2()
    UserForm2.Show vbModeless
    AppActivate ("Microsoft Excel")
End Sub
```

DISABLING THE USERFORM'S CLOSE BUTTON

Some of your UserForms might require input before the user can proceed further. To enforce user input, you can disable the Close button, usually located at the far right of the UserForm's title bar. This is not an everyday happenstance but when your project requires input at a critical point in a process, you need a way to keep the UserForm active until the required information is input.

UserForms have a QueryClose event that can help you control such situations. In Figure 23-2, a message box appears if the "X" Close button is clicked in an attempt to close the UserForm without selecting a name from the drop-down list. The code associated with that follows Figure 23-2.

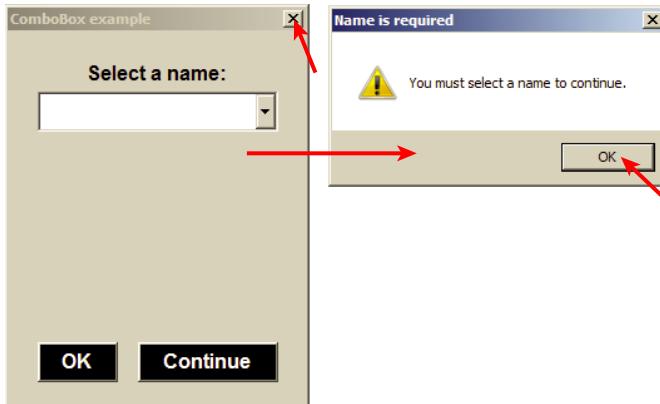


FIGURE 23-2

```
Private Sub UserForm_QueryClose(Cancel As Integer, CloseMode As Integer)
    'Prevents use of the Close button if a name has not been selected.
    If CloseMode = vbFormControlMenu And Len(ComboBox1.Value) = 0 Then
        Cancel = True
        MsgBox "You must select a name to continue.", , "Name is required"
        'Set Focus to the ComboBox for the user.
        ComboBox1.SetFocus
    End If
End Sub
```

Keep in mind that you want to monitor the input requirement through the other controls on the UserForm as well. The following example is associated with the Continue button:

```
Private Sub cmdContinue_Click()
If Len(ComboBox1.Value) = 0 Then
    MsgBox "You must select a name to continue.", , "Name is required"
    'Set Focus to the ComboBox for the user.
    ComboBox1.SetFocus
    Exit Sub
Else
    Unload Me
End If
End Sub
```

MAXIMIZING YOUR USERFORM'S SIZE

If you want to fill the screen with just your UserForm and nothing else, the following code in the Initialize event can help you do that. Be aware that some adjustment to the code might be needed with the Zoom property, in case the UserForm is so small to begin with that its fully expanded size exceeds the window's Zoom capacity.

```
Private Sub UserForm_Initialize()
With Application
    .WindowState = xlMaximized
    Zoom = Int(.Width / Me.Width * 100)
    Width = .Width
    Height = .Height
End With
End Sub
```

NOTE You don't need to settle for the UserForm loading in the center of your screen. You can specify the location, such as with the following example that shows the UserForm in the top-left corner of the screen:

```
Private Sub UserForm_Initialize()
Me.StartUpPosition = 0
Me.Top = Application.Top
Me.Left = Application.Left
End Sub
```

SELECTING AND DISPLAYING PHOTOGRAPHS ON A USERFORM

An Image control helps you display a graphic object, such as a picture, on a UserForm. You have three ways to place a picture onto an Image control—two are manual methods and one is a VBA method.

Suppose you have a picture file on your computer, such as your company's logo, that you want to show for a customized look on your UserForm. You can use VBA's LoadPicture method to load the picture file onto the Image control when you call the UserForm, with the following example:

```
Private Sub UserForm_Initialize()
    Image1.Picture = LoadPicture("C:\CompanyPictures\CompanyLogo.jpg")
End Sub
```

This method works great, so long as the picture file exists in that folder path for every computer on which the UserForm will ever be opened, which is not likely. As you develop UserForms for others' use outside a shared network environment, you want to load a picture onto an Image control manually, and forego the VBA route.

You can load an Image control manually in two ways. In the UserForm's design window, place the Image control where you want it on the UserForm. Activate the Image control's Properties window and locate the Picture property. Placing your cursor inside the Picture property exposes a small ellipsis button, as shown in Figure 23-3. Click that button to show the Load Picture dialog box. From the Load Picture dialog box, navigate to the picture file you want to load, select it, and click Open.

The other manual alternative is even simpler. After you've added your Image control, select your picture object and press Ctrl+C to place it on the clipboard. Select the Image control on the UserForm, select its Picture property in the Properties window, click inside the Picture property, and press Ctrl+V to paste the picture into the Image control.

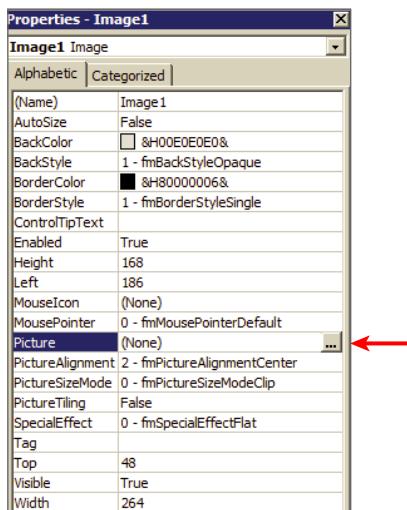


FIGURE 23-3

UNLOADING A USERFORM AUTOMATICALLY

Have you ever wanted to show a UserForm for a limited period of time, and then unload it without user intervention? UserForms need not serve the sole purpose of user input. Sometimes they can be opportunistically employed as a mechanism for a specialized greeting, or, if tastefully designed, an informative splash screen.

Personally, I do not appreciate most of the splash screens I see when opening various software applications; many look like cheap advertisements that waste the user's time. However, a nice opening welcome message to customize the look and feel of your workbook can be a good thing if designed well, but do keep the visible time to a maximum of five seconds; any longer than that is an annoyance.

Call the UserForm as you normally would. The following code goes into the UserForm module, in this example for a five-second appearance:

```
Private Sub UserForm_Activate()
Application.Wait (Now + TimeValue("0:00:05"))
Unload Me
End Sub
```

PRE-SORTING THE LISTBOX AND COMBOBOX ITEMS

Suppose you want to import a list of items into your ListBox (or ComboBox) such as a list of cities in range A1:A20 as shown in Figure 23-4. You can do that easily with this event code for a ListBox:

```
Private Sub UserForm_Initialize()
ListBox1.List = Range("A1:A20").Value
End Sub
```

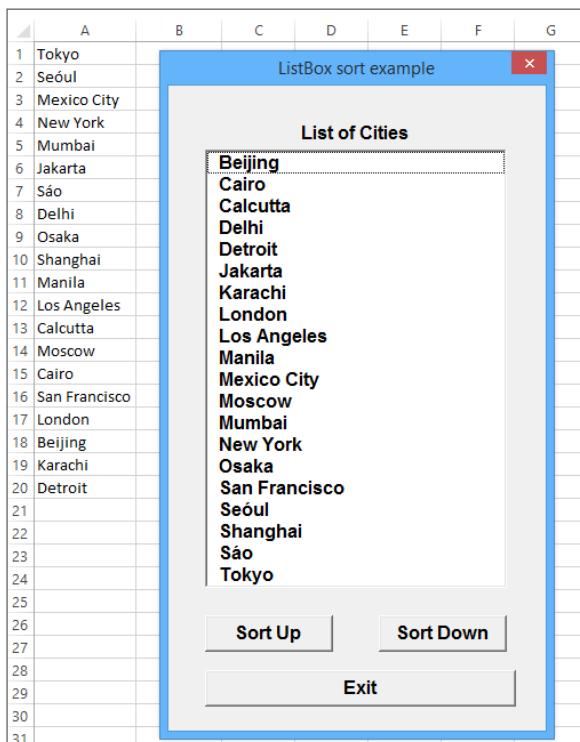


FIGURE 23-4

Lists tend to be easier to work with when they are alphabetized. To handle that seamlessly for the user, the following amendment to the preceding code is a series of loops with variables that examine each element in the ListBox, and sorts it in ascending alphabetical order. The result is shown in Figure 23-4.

```
Private Sub UserForm_Initialize()
ListBox1.List = Range("A1:A20").Value
Dim x As Integer, y As Integer, z As String
With ListBox1
For x = 0 To .ListCount - 2
For y = x + 1 To .ListCount - 1
If .List(x) > .List(y) Then
z = .List(y)
.List(y) = .List(x)
.List(x) = z
End If
Next y
Next x
End With
End Sub
```

Notice two additional CommandButtons near the bottom of the UserForm. One is captioned Sort Up and the other is captioned Sort Down. Users appreciate the ability to customize the look of their interface. If it is easier for some people to read a list from Z to A, and others from A to Z, so be it. The following code shows an example of how each button, when clicked, sorts the ListBox. First, ascending:

```
Private Sub cmdSortUp_Click()
Dim x As Integer, y As Integer, z As String
'Sort ascending
With ListBox1
For x = 0 To .ListCount - 2
For y = x + 1 To .ListCount - 1
If .List(x) > .List(y) Then
z = .List(y)
.List(y) = .List(x)
.List(x) = z
End If
Next y
Next x
End With
End Sub
```

Then, descending:

```
Private Sub cmdSortDown_Click()
Dim x As Integer, y As Integer, z As String
'Sort descending
With ListBox1
For x = 0 To .ListCount - 2
For y = x + 1 To .ListCount - 1
If .List(x) < .List(y) Then
```

```

z = .List(y)
.List(y) = .List(x)
.List(x) = z
End If
Next y
Next x
End With
End Sub

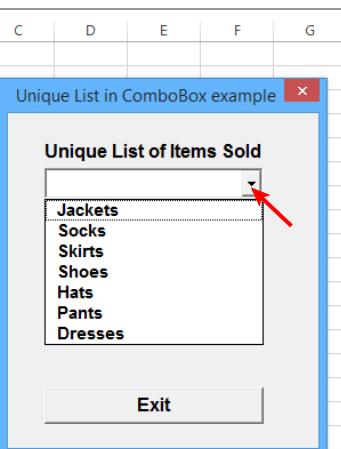
```

NOTE If you were to do this in real practice, you'd eliminate the redundancy of declaring the same variables for each event, and instead publicly declare them once.

POPULATING LISTBOXES AND COMBOBOXES WITH UNIQUE ITEMS

As often as not, when you load a ListBox or ComboBox with a source list of items from a worksheet, the range is dynamic, meaning the length of the list varies. Also, chances are pretty good that the source list contains duplicate entries, and there is no need to place more than one unique item in a ListBox or ComboBox.

In Figure 23-5, column A contains a list of clothing items that were sold in a department store. A unique list of these items was compiled in a ComboBox as shown in Figure 23-5, with the following code to demonstrate how to populate the ComboBox in this manner when the length of the source list is not known, and some cells in the source list might have no entry.



A	B	C	D	E	F	G
1 Items Sold	How many sold					
2 Jackets	32					
3 Socks	37					
4 Skirts	53					
5 Jackets	17					
6 Shoes	31					
7 Socks	23					
8 Hats	72					
9 Shoes	49					
10 Jackets	22					
11 Pants	67					
12 Socks	73					
13 Hats	40					
14 Pants	89					
15 Hats	96					
16 Skirts	98					
17 Dresses	30					
18 Pants	55					
19 Dresses	10					
20 Skirts	99					
21 Socks	65					
22 Dresses	63					
23 Socks	33					
24 Jackets	62					
25 Shoes	71					
26						

FIGURE 23-5

```

Private Sub UserForm_Initialize()
'Declare variables for a Collection and cell range.
Dim myCollection As Collection, cell As Range

'Error bypass to set a new collection.
On Error Resume Next
Set myCollection = New Collection

'Open a With structure for the ComboBox
With ComboBox1
'Clear the ComboBox
.Clear

'Open a For Next loop to examine every cell starting with A2
'and down to the last used cell in column A.
For Each cell In Range("A2:A" & Cells(Rows.Count, 1).End(xlUp).Row)

'If the cell is not blank...
If Len(cell) <> 0 Then
'Clear the possible error for a Collection
'possibly not having been established yet.
Err.Clear
'Add the cell's value to the Collection.
myCollection.Add cell.Value, cell.Value
'If there is no error, that is, if the value does not
'already exist in the Collection, add the item to the ComboBox.
If Err.Number = 0 Then .AddItem cell.Value
End If

'Loop to the next cell.
Next cell

'Close the With structure for the ComboBox.
End With
End Sub

```

NOTE If you want the first item in the ComboBox's list to be visible when the UserForm is called, add this line before the End Sub line:

```
ComboBox1.ListIndex = 0
```

To expand a bit on the possible usefulness of listing unique items in a ComboBox, see the example in Figure 23-6, where two Label controls were added (named Label2 and Label3) to the right of the ComboBox. When the ComboBox value is changed with the following code, Label2's caption reflects the value item, and Label3's caption sums the items sold in column B for the item that was selected in the ComboBox.

```

Private Sub ComboBox1_Change()
Label2.Caption =
"Total " & ComboBox1.Value & " Sold:"
Label3.Caption =
WorksheetFunction.SumIf(Columns(1), ComboBox1.Value, Columns(2))
End Sub

```

	A	B	C	D	E	F	G	H	I	J
1	Items Sold	How many sold								
2	Jackets	32								
3	Socks	37								
4	Skirts	53								
5	Jackets	17								
6	Shoes	31								
7	Socks	23								
8	Hats	72								
9	Shoes	49								
10	Jackets	22								
11	Pants	67								
12	Socks	73								
13	Hats	40								
14	Pants	89								
15	Hats	96								
16	Skirts	98								
17	Dresses	30								
18	Pants	55								
19	Dresses	10								
20	Skirts	99								
21	Socks	65								
22	Dresses	63								
23	Socks	33								
24	Jackets	62								
25	Shoes	71								
26										



FIGURE 23-6

DISPLAYING A REAL-TIME CHART IN A USERFORM

Earlier in this lesson you saw how to load a picture into an Image control. You can also create a temporary graphic file on the fly, load that file into a UserForm's Image control, and delete the temporary graphic file, all with the user being none the wiser.

Figure 23-7 shows a list of cities, ranked by their approximate population. Elsewhere in the workbook is a chart sheet named Chart1 with a bar chart of this city population data. You can represent the Chart1 sheet's chart in real time by exporting its image as a .gif file and loading it onto an Image control when the UserForm is called. Figure 23-7 shows the result and following that is the Initialize event code that handles this task.

```
Private Sub UserForm_Initialize()
    ActiveWorkbook.Charts("Chart1").Export "CityPopulation.gif"
    Image1.Picture = LoadPicture("CityPopulation.gif")
    Image1.PictureSizeMode = fmPictureSizeModeZoom
    Kill "CityPopulation.gif"
End Sub
```

NOTE You can print a UserForm, even if it is not open, with the following line:

```
UserForm1.PrintForm
```

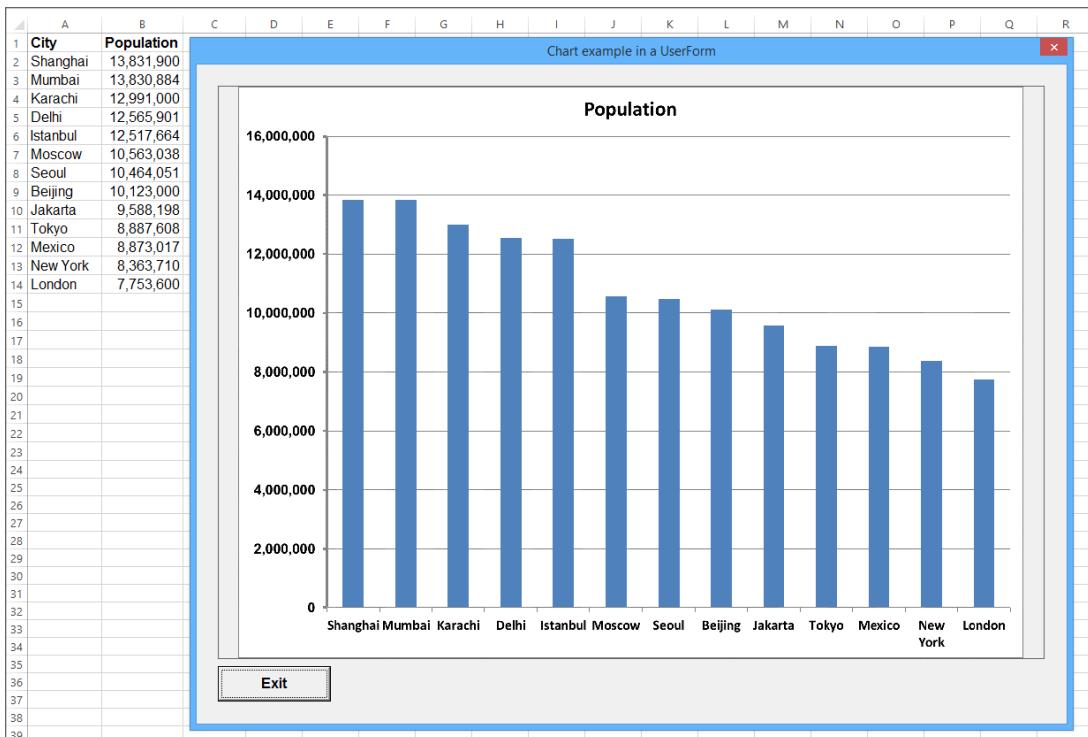


FIGURE 23-7

TRY IT

For this lesson, you design a UserForm to have the basic functionality of a web browser, including the ability to navigate to the websites of your choice, go backward and forward to websites, and set the initial website when the UserForm is initialized.

Lesson Requirements

To get the workbook, you can download Lesson 23 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open a new workbook and press Alt+F11 to get into the Visual Basic Editor.
2. If the Project Explorer window is not visible, press Ctrl+R, and if the Properties window is not visible, press the F4 key.
3. In the Project Explorer window, select your workbook name, and from the menu bar click Insert ⇔ UserForm.
4. In the Properties window for that UserForm, accept the default Name property of UserForm1, set the Height property to 540 and the Width property to 852.

5. Click the Toolbox icon, or from the menu bar click View \Rightarrow Toolbox.
6. Draw a TextBox near the upper-left corner of the UserForm. Accept the default Name property of TextBox1, set its Height property to 24, and its Width property to 252.
7. Draw four CommandButtons along the top of the UserForm to the right of the TextBox. Each CommandButton should be the same size, with its Height property set at 24 and its Width property set at 120.
8. Name the first CommandButton cmdNavigate and label its Caption property as Navigate. Set its Default property to True.
9. Name the second CommandButton cmdBack and label its Caption property as Back.
10. Name the third CommandButton cmdForward and label its Caption property as Forward.
11. Name the fourth CommandButton cmdExit and label its Caption property as Exit.
12. The final control is a WebBrowser, and chances are its icon is not on your Toolbox's Cover tab. If that's the case, right-click the Cover tab and select Additional Controls as shown in Figure 23-8.

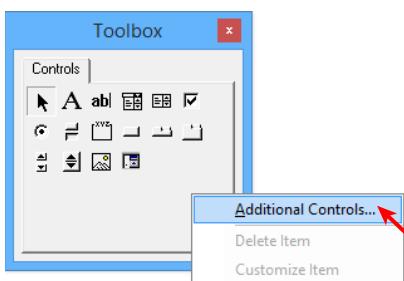


FIGURE 23-8

13. Scroll down the list of available controls and select Microsoft Web Browser as shown in Figure 23-9. Click OK to place the WebBrowser icon on your Toolbox's Cover tab as shown in the lower-left corner of Figure 23-10.

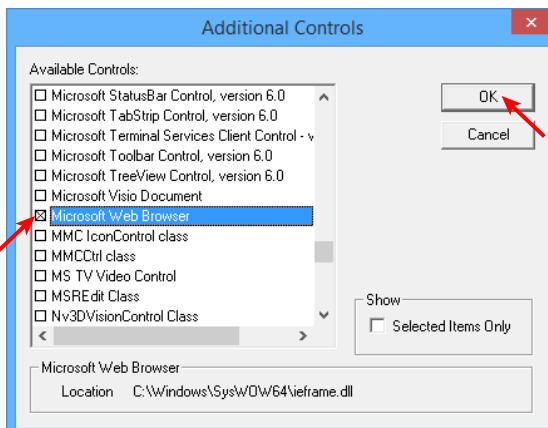


FIGURE 23-9



FIGURE 23-10

14. Click to select the WebBrowser icon on the Toolbox just as you would with any control, and draw a WebBrowser control onto the open area of the UserForm. Accept the default Name property of WebBrowser1, and then set its Height property to 450 and its Width property to 816. This completes the design of the UserForm, which in the VBE looks like Figure 23-11.

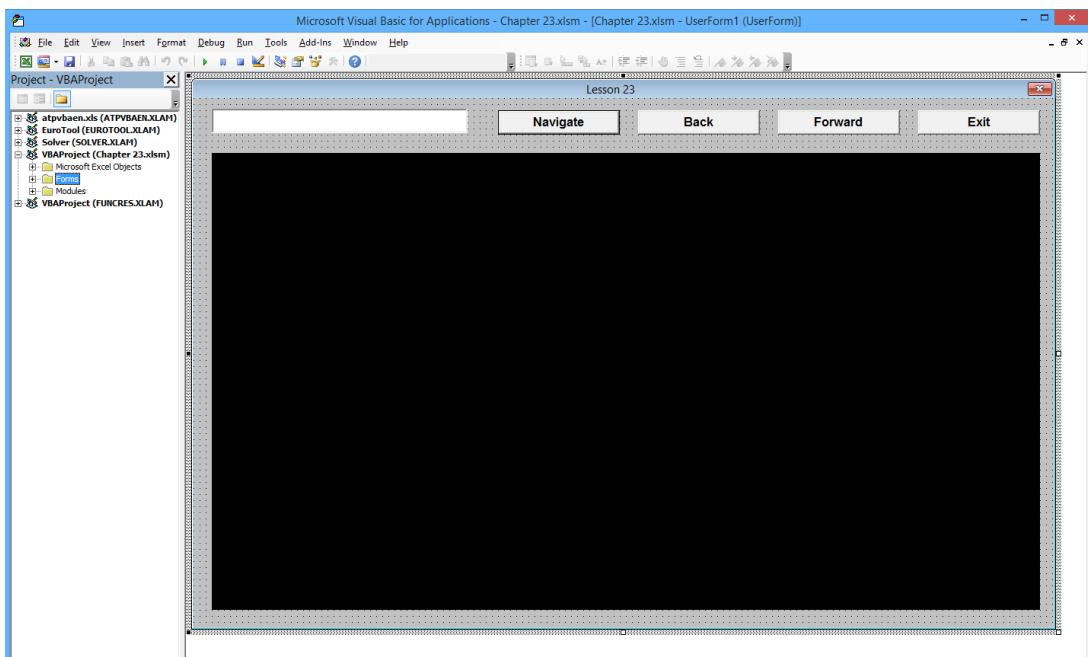


FIGURE 23-11

15. The code associated with this UserForm is surprisingly simple. Double-click the UserForm to access its module. In the Object drop-down list, select UserForm and in the Procedure drop-down list select Initialize. The Initialize event is a single line of code that tells the WebBrowser which website to navigate to when the UserForm initializes, similar to the homepage setting on your web browser. In this example, I entered the website for Microsoft, at www.microsoft.com. Here is the entire Initialize event with that navigation command:

```
Private Sub UserForm_Initialize()
    WebBrowser1.Navigate "http://www.microsoft.com"
End Sub
```

16. You have an Exit button named cmdExit, so use the Unload Me command for that:

```
Private Sub cmdExit_Click()
Unload Me
End Sub
```

17. Regarding the CommandButton for navigation, the process starts by the user entering a website address in the TextBox. The user can then either click the cmdNavigate button, or press the Enter key because you set the Default property to True for the cmdNavigate button in Step 8. Thinking ahead for more convenience, you can structure the cmdNavigate's Click event to assume that all web addresses start with "http://www." which saves the user time and effort by just entering the web address's domain name. For example, instead of entering http://www.somewhere.com in the TextBox, a user need only enter somewhere.com with this code for the cmdNavigate button:

```
Private Sub cmdNavigate_Click()
WebBrowser1.Navigate "http://www." & TextBox1.Text
End Sub
```

18. All that's left are the two buttons for Back and Forward, easily handled with the WebBrowser control's GoBack and GoForward methods. For both methods, On Error Resume Next is utilized to avoid a possible runtime error if the browsing session is at its starting or ending point when the cmdBack or cmdForward button is clicked. Here is the code for the Back CommandButton:

```
Private Sub cmdBack_Click()
On Error Resume Next
WebBrowser1.GoBack
Err.Clear
End Sub
```

Here is the code for the Forward CommandButton:

```
Private Sub cmdForward_Click()
On Error Resume Next
WebBrowser1.GoForward
Err.Clear
End Sub
```

19. When you call the UserForm, Figure 23-12 shows an example that is similar to what you see.

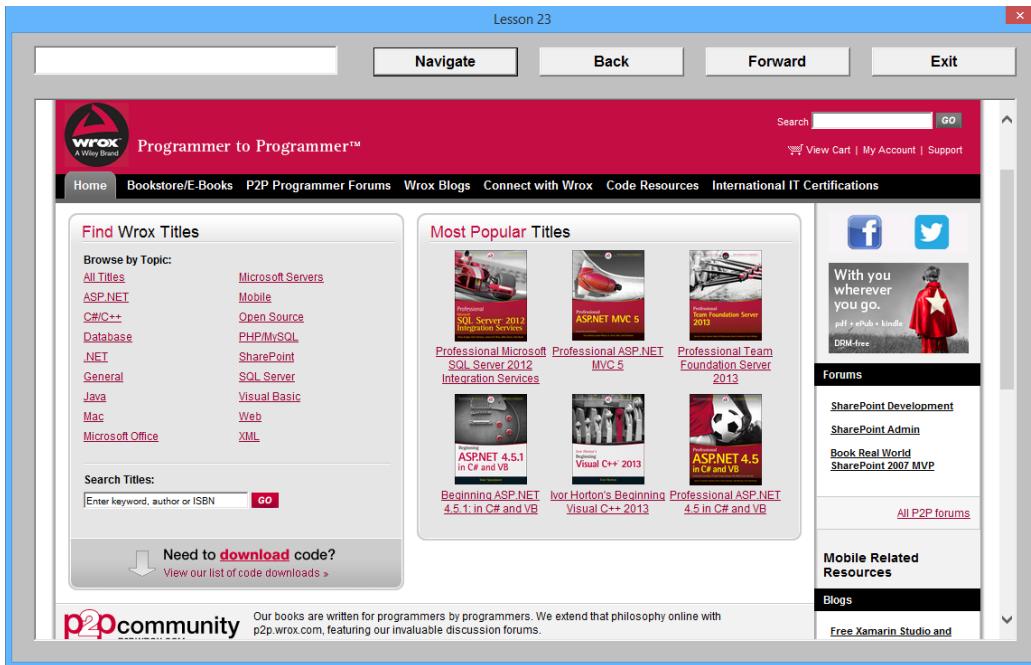


FIGURE 23-12

REFERENCE Please select the video for Lesson 23 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

24

Class Modules

Class modules—the very name has caused many a burgeoning Excel VBA programmer to turn toward other areas of VBA study. For some reason, the use of class modules is not a skill held by many otherwise knowledgeable VBA programmers, despite the power and flexibility class modules can provide to your workbook projects.

Class modules are not rocket science, but they are a different kind of VBA animal that takes some extra attention to grasp. I want to express three objectives in this lesson:

- Explain what classes and class modules are.
- Describe what class modules can do for you.
- Provide examples of class modules applied to UserForm and embedded worksheet controls.

Here is an opportunity for you to set yourself apart from the VBA crowd and learn a valuable skill that has actually been available in Excel since Office 97. Though you won't need class modules for most of your projects, this lesson helps you recognize when the time is right to use class modules, and most importantly, how to program them.

WHAT IS A CLASS?

A *class* is the formalized definition of an object that you create. Your first reaction might be to wonder why you'd ever need to create yet another object in Excel, which seemingly has no shortage of objects. Actually, you normally don't *need* to, but there will be times when your workbook will be better off if you do.

A new class (as in classification) is like a blueprint for your created object and its properties, methods, and events. In Lesson 19 you learned about user-defined functions; where class modules are concerned, you can think of a class as a user-defined model for an object that you create. You see examples later in the lesson that help clarify the theory.

NOTE It's easy to get lost on any new topic if the emphasis on learning it is based on definitions and theory. That is why most of this lesson relies on real-world examples to show what class modules are all about. Though kept to a minimum, the definitions and theory in this lesson are useful for you to gain a perspective on class modules. If you don't fully comprehend all definitions the first time around, don't worry—the VBA examples will be your biggest ally in helping you understand the process of developing class modules.

WHAT IS A CLASS MODULE?

A *class module* is a special module in the Visual Basic Editor whose purpose is to hold VBA code that defines classes. A class module looks like any other kind of module you have seen, and in its own way acts like one, too. For example, whereas the code for worksheet event procedures goes into worksheet modules, the code for creating and defining classes goes into class modules.

You create a class module in the VBE by choosing *Insert* → *Class Module* from the menu bar as shown in Figure 24-1. A class module is created with the default name of Class1 as shown in Figure 24-2.

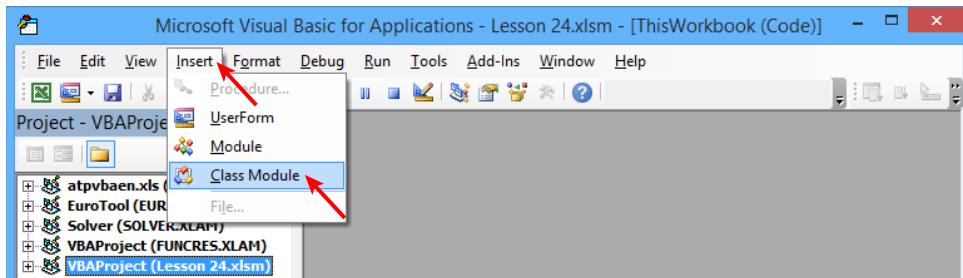


FIGURE 24-1

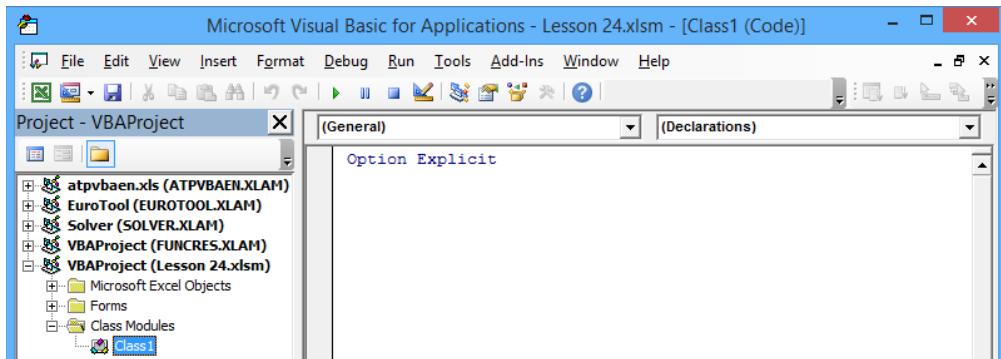


FIGURE 24-2

NOTE There is a one-to-one relationship between a class and a class module. A class module provides for only one class to be defined. If you need to define three classes in your workbook, you need three class modules, one for each class. For example, suppose you have several CheckBox controls on your UserForm, and you want to color the CheckBoxes green when they are checked and red when they are unchecked. Instead of coding this functionality for every CheckBox's Click event, you can use a class module that groups all the CheckBoxes as a single collection object. That way, all CheckBoxes respond to the same Click event, with one VBA class procedure. If you also want some (or all) of the CommandButtons on a UserForm in that same workbook to respond to, say, a MouseMove event, you'd create another class module for that.

CREATING YOUR OWN OBJECTS

I started this lesson saying that many VBA programmers have avoided the topic of class modules, and it wouldn't surprise me if a primary culprit is VBA's intentionally vague concept of class objects. Seeing actual VBA examples of class modules in everyday situations is the best way to pick up the concept of class objects.

Here's the theoretical synopsis: A class is defined in a class module, and you can think of a class as a blueprint or template for an object. In the context of class modules, the term *object* can be almost any object in Excel whose functionality you want to expand. This concept becomes clearer with VBA examples in this lesson that deal with controls that are embedded in a worksheet or are placed onto UserForms. You can have those controls all respond to one single event, instead of needing to write numerous redundant procedures for each control.

A class module only serves the purpose of holding the code that defines (but does not create) a class object. In some other module that is not a class module, such as a UserForm module or workbook module (depending on the task you are solving), you can declare a variable of the class type and create an instance of that class (known as *instantiating* the class) with the New keyword. Upon instantiation, your declared variable becomes an object whose events, properties, and methods are defined by your code in the class module.

AN IMPORTANT BENEFIT OF CLASS MODULES

Suppose you have a UserForm with 12 TextBoxes, into which a dollar figure for budgeted expenses is to be entered for each month of the year, as in the example shown in Figure 24-3.

It's important that only numbers are entered, so you want to validate every TextBox entry to be numeric, while disallowing entry of an alphabetic letter, symbol, or any character other than a number. The following example can handle that for TextBox1 in the UserForm module:

```
Private Sub TextBox1_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
Select Case KeyAscii
Case 48 To 57 'numbers 0-9
Case Else
```

```

KeyAscii = 0
MsgBox "You made a non-numeric entry.", vbCritical, "Numbers only please."
End Select
End Sub

```

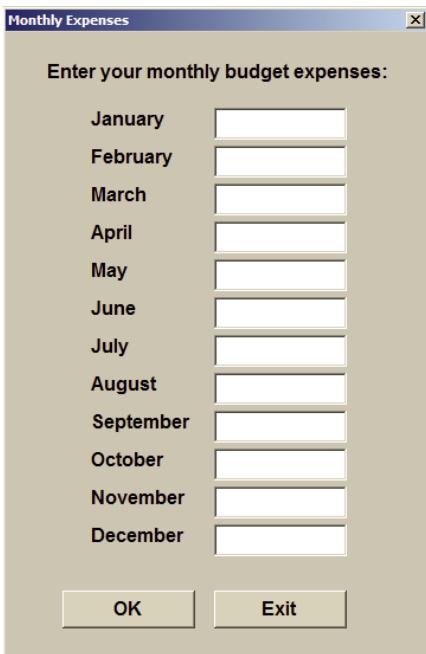


FIGURE 24-3

You can maybe get away with the redundancy of writing 12 separate events to monitor the entries in each TextBox. But what happens if your project requires 100 TextBoxes, or if the numeric validation process expands to allow decimals or negative numbers? You'd have to do a lot of updates for each TextBox, and the volume of redundant code creates a bad design that's destined for human error and runtime failure.

If you insert a class module instead, you can define an object that would be a group of 12 TextBoxes. You can name your group object `TxtGroup` and indicate that the objects in that group are `TextBoxes`. There is nothing special about the name `TxtGroup`. I chose it because the idea is to group TextBoxes, but whatever object name makes sense to you works just as well.

The following VBA declaration statement is a common example that is placed at the top of your class module. It defines the class object and includes the `WithEvents` keyword, which exposes the events associated with TextBoxes:

```
Public WithEvents TxtGroup As MSForms.TextBox
```

Now that you have established the `TxtGroup` object as a group of TextBoxes, you can invoke it to handle the same `KeyPress` event that you might have written individually for all 12 TextBoxes. As shown in the following code, you now make the `TxtGroup` object recognize the `KeyPress` event triggered by keyboard data entry upon any one of its 12 TextBoxes. The code to handle an event for all 12 TextBoxes is the same for `TxtGroup` as it is for `TextBox1`, except for the name of the object:

```

Private Sub TxtGroup_KeyPress(ByVal KeyAscii As MSForms.ReturnInteger)
Select Case KeyAscii
Case 48 To 57 'numbers 0-9
Case Else
KeyAscii = 0
MsgBox "You made a non-numeric entry.", vbCritical, "Numbers only please."
End Select
End Sub

```

Keep in mind that, so far, all you have done is define the object, but it still exists only as a concept. The next step is to create your defined object (formally known as instantiating it) to make it a working object that responds to events, and becomes associated with methods and properties. At this moment, with the UserForm created and the class module selected with the preceding code in it, your work in the class module is complete. Your VBE window should look similar to Figure 24-4.

The final step is to go into the UserForm module and instantiate the `TxtGroup` object that is a group of 12 TextBoxes. At the top of the UserForm module, declare a variable for 12 TextBoxes to instantiate the `TxtGroup` class object, with the `New` keyword for the `Class1` module name:

```
Dim txtBoxes(1 To 12) As New Class1
```

Using the `Initialize` event, declare an `Integer` type variable that assists in looping through the 12 TextBoxes. Set each TextBox as a member of the `TxtGroup` class:

```

Private Sub UserForm_Initialize()
Dim intCounterTextBox As Integer
For intCounterTextBox = 1 To 12
Set txtBoxes(intCounterTextBox).TxtGroup = _
Controls("TextBox" & intCounterTextBox)
Next intCounterTextBox
End Sub

```

Your entire coding process relating to the class module is complete, and it is quite a bit shorter than all the code you'd have amassed if you coded the `KeyPress` event for every TextBox! If you were to open the UserForm and attempt a non-numeric character in any of the 12 TextBoxes, that character would be disallowed and the message box would appear, looking like Figure 24-5.

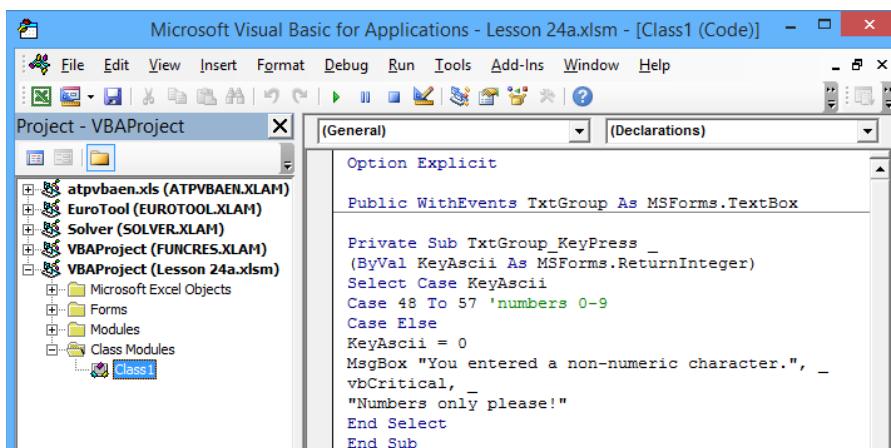


FIGURE 24-4

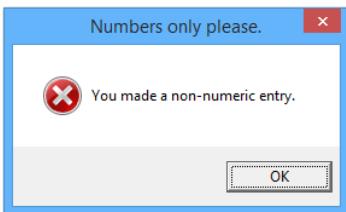


FIGURE 24-5

CREATING COLLECTIONS

In the preceding example, you created a class for 12 TextBoxes. You knew ahead of time the number of TextBoxes was 12 because there was a TextBox for each of the 12 calendar months. The question becomes, what do you do if the count of inclusive TextBoxes is not known? What if your project is so wide in scope that TextBoxes are being frequently added and subtracted from the UserForm, and you don't want to keep modifying the code with every change in TextBox count?

The answer is to create a *collection* of TextBoxes by looping through all the controls in the UserForm. Then, when a TextBox is encountered in the loop, it is automatically added to the collection, which is then transferred to the class object. Assuming the event code you placed in the class module has not changed, all you need to adjust is the code in the UserForm module using the previous example. The first item of business is to prepare a declaration statement at the top of the module that does not specify a count of TextBox names, such as the following example:

```
Dim TxtGroup() As New Class1
```

Next, the following code in the UserForm's Initialize event wraps up all the TextBoxes into one array package using the ReDim Preserve keywords. This method does not depend on how many TextBoxes are present on the UserForm it simply collects all the ones into the `TxtGroup` object that it finds:

```
Private Sub UserForm_Initialize()
Dim intCounterTextBox As Integer, ctl As Control
intCounterTextBox = 0
For Each ctl In Controls
If TypeName(ctl) = "TextBox" Then
intCounterTextBox = intCounterTextBox + 1
ReDim Preserve TxtGroup(1 To intCounterTextBox)
Set TxtGroup(intCounterTextBox).TxtGroup = ctl
End If
Next ctl
End Sub
```

CLASS MODULES FOR EMBEDDED OBJECTS

So far, UserForms have been the backdrop for objects in a class module. You can also create a class of objects embedded on worksheets, such as charts, pivot tables, and ActiveX controls. In the case of ActiveX controls, it's worth mentioning a syntax difference when referring to them.

Suppose you have an unknown number of CommandButtons on Sheet1 and you want to create a class module to determine which button was clicked, without having to program every CommandButton's Click event. This example of code in a class module named Class1 demonstrates how to extract the

name, caption, and address of the cell being touched by the top-left corner of the CommandButton object. Figure 24-6 shows the message box that appears when you click one of the CommandButtons.

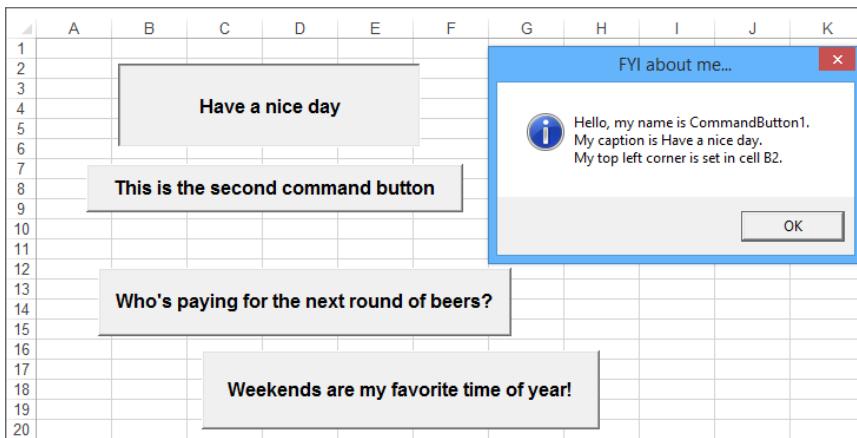


FIGURE 24-6

```
Public WithEvents cmdButtonGroup As CommandButton
Private Sub cmdButtonGroup_Click()
    MsgBox _
        "Hello, my name is '" & _
        cmdButtonGroup.Name & "'." & vbCrLf & _
        "My caption is '" & _
        cmdButtonGroup.Caption & "'." & vbCrLf & _
        "My top left corner is set in cell " & _
        cmdButtonGroup.TopLeftCell.Address(0, 0) & ".", _
        64, "You just clicked me, here's my info :"
End Sub
```

You can also tap into other events in the same class module. All that's required is that you use the same class object (cmdButtonGroup in this example), and that the event is supported by the object. With CommandButtons, the MouseOver event can help you identify which button you are hovering your mouse over by shading it orange, while all other CommandButtons on the sheet are colored gray.

NOTE I used hex codes in this example for the buttons' BackColor property, to show how you'd use hex in code to refer to colors. These hex values are always shown in the Properties window of ActiveX controls for BackColor and ForeColor properties, and I personally find them very reliable in VBA code with any version of Excel.

```
Private Sub cmdButtonGroup_MouseMove(ByVal Button As Integer, _
    ByVal Shift As Integer, ByVal X As Single, ByVal Y As Single)
    Dim myBtn As Object
    For Each myBtn In ActiveSheet.OLEObjects
        If TypeName(myBtn.Object) = "CommandButton" Then _
            myBtn.Object.BackColor = &HC0C0C0 'turn all to gray
    Next myBtn
    cmdButtonGroup.BackColor = &H80FF& 'orange
End Sub
```

NOTE As you can probably tell, despite the appearance of differently shaped CommandButtons with comical captions, the larger point of this example is that you can capture various properties of class objects, assign them to a variable, and utilize that variable information in other macros, or even as part of the class module's event code. For example, in real practice, you don't need or want a message box to pop up and tell you which button you just clicked; you already know that. If, for example, your project is such that the CommandButtons' captions have a word or phrase to be used as a criterion for automatically filtering a table of data, this application of flexible class module coding will save you a lot of work.

For embedded ActiveX controls, you can instantiate the collection of OLE objects, in this example for CommandButtons, with the following code that goes into the ThisWorkbook module. Be sure to place this example declaration statement at the top of the ThisWorkbook module:

```
Dim cmdButtonHandler() As New Class1
```

Finally, utilize the Open event to collect the CommandButtons that are only on Sheet1. Notice the references to the OLEObject and OLEObjects keywords when dealing with embedded ActiveX controls:

```
Private Sub Workbook_Open()
Dim cmdButtonQuantity As Integer, MYcmdButton As OLEObject
cmdButtonQuantity = 0
With ThisWorkbook
For Each MYcmdButton In .Worksheets("Sheet1").OLEObjects
If TypeName(MYcmdButton.Object) = "CommandButton" Then
cmdButtonQuantity = cmdButtonQuantity + 1
ReDim Preserve cmdButtonHandler(1 To cmdButtonQuantity)
Set cmdButtonHandler(cmdButtonQuantity).cmdButtonGroup =
= MYcmdButton.Object
End If
Next MYcmdButton
End With
End Sub
```

Not all controls recognize the same event types, though, so you'd need to set a class event that the object type can recognize.

There is another technique using the Collection keyword for grouping the same types of objects into a class. In this example, Sheet1 has a number of embedded CheckBox controls, and you want to write one small piece of VBA code that applies to all CheckBoxes.

The visual effect you want is for any CheckBox on Sheet1 to be shaded black if it is checked, and white if it is unchecked. Figure 24-7 shows the differences in color shading depending on the status of the CheckBoxes.

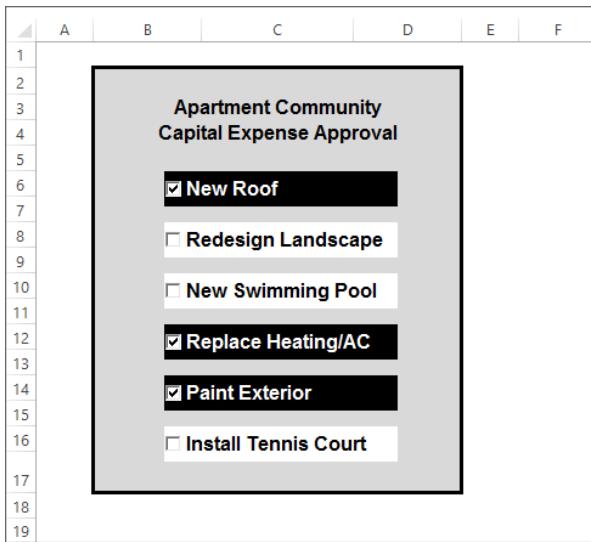


FIGURE 24-7

The code to do this is surprisingly minimal. Insert a new class module, and assuming it is named `Class2` because you already have a `Class1` module established, this code goes into the `Class2` module:

```
Public WithEvents grpCBX As MSForms.CheckBox

Private Sub grpCBX_Click()
With grpCBX

If .Value = True Then
    .BackColor = &H0& 'Black background
    .ForeColor = &HFFFFFF 'White font

Else

    .BackColor = &HFFFFFF 'White background
    .ForeColor = &H0& 'Black font

End If

End With
End Sub
```

The rest of the code goes into the `ThisWorkbook` module. It instantiates the `grpCBX` object and is refreshed each time the workbook opens by utilizing the `Workbook_Open` event:

```
Public myControls As Collection
Private Sub Workbook_Open()
Dim oleCtl As OLEObject, ctl As Class2
Set myControls = New Collection
For Each oleCtl In Worksheets("Sheet1").OLEObjects
If TypeOf oleCtl.Object Is MSForms.CheckBox Then
    Set ctl = New Class1
```

```
Set ctl.grpCBX = oleCtl.Object  
myControls.Add ctl  
End If  
Next  
End Sub
```

TRY IT

For this lesson, you create a class module to handle the `Click` event of some of the OptionButtons on a UserForm. You design a simple UserForm with eight OptionButtons, of which only five are a part of the class module that identifies by name which OptionButton and caption was clicked.

Lesson Requirements

To get the sample workbook, you can download Lesson 24 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open a new workbook.
2. Press Alt+F11 to get into the Visual Basic Editor.
3. From the menu bar, click Insert \Rightarrow UserForm, and size the UserForm to a Height of 200 and a Width of 400.
4. Draw a Label control near the top-left corner of your UserForm, and caption it as `OptionButtons In Class Module`.
5. Draw a Label control near the top-right corner of your UserForm and caption it as `Other OptionButtons`. Figure 24-8 shows how your UserForm should look so far.

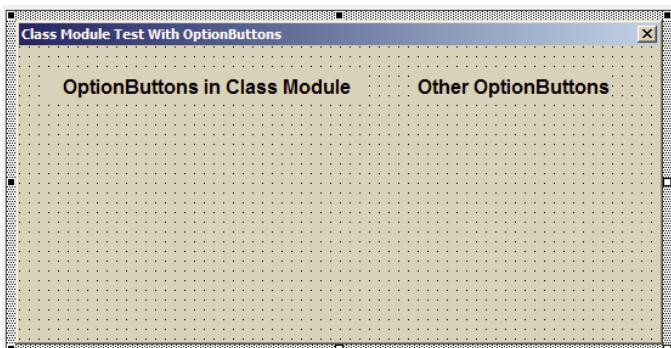


FIGURE 24-8

6. Under the first Label control, draw a vertical column of five OptionButtons. A fast way to do this is to draw one OptionButton and then copy and paste it four times. Change the captions of those five OptionButtons to **Apples**, **Bananas**, **Peaches**, **Grapes**, and **Oranges**, as shown in Figure 24-9.

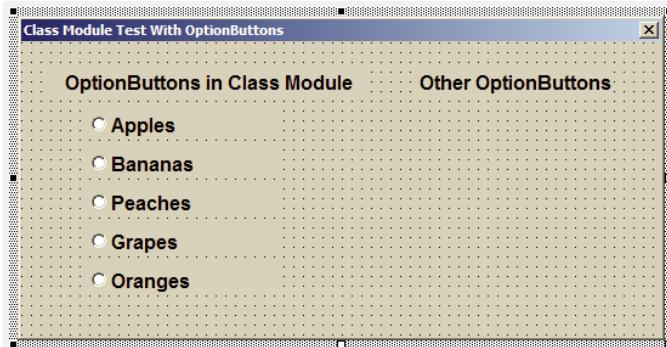


FIGURE 24-9

7. Paste three more OptionButtons below the second Label control. Change the captions of those three OptionButtons to **Plums**, **Pears**, and **Tangerines**. You now have eight OptionButtons on your UserForm, all with different captions that are the names of fruits. The actual VBA names of the eight OptionButtons have not changed; they all are still named by default as OptionButton1, OptionButton2, and so on, to OptionButton8. For example, if you were to select the OptionButton that is captioned Oranges, you would see in its Properties window that it is named OptionButton5. Figure 24-10 shows how your UserForm looks at this point.

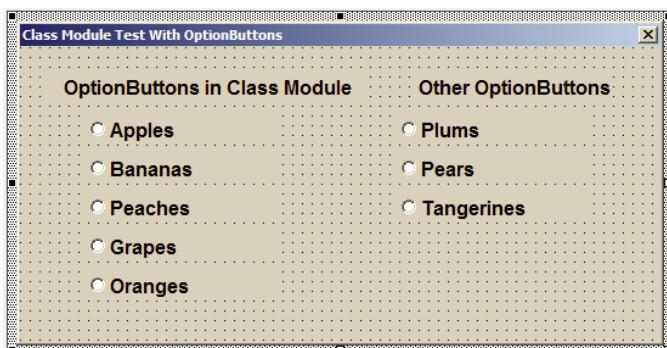


FIGURE 24-10

8. Draw a CommandButton in the lower-right corner of the UserForm. Name it **cmdExit** and caption it as **Exit**.

9. Double-click the cmdExit button, which takes you into the UserForm's module, with the cmdExit button's Click event ready for your code. Type `Unload Me`, and your UserForm module in the VBE looks like Figure 24-11.

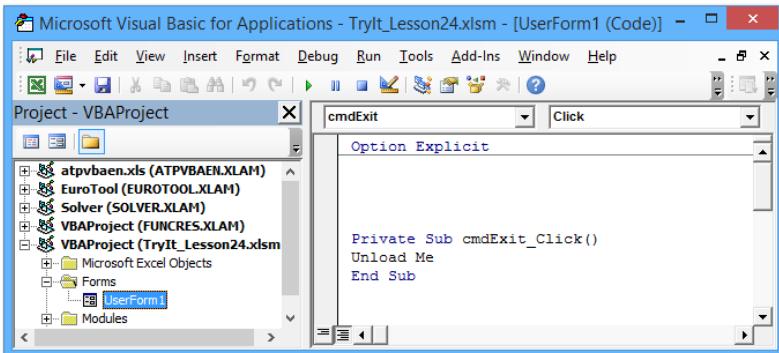


FIGURE 24-11

10. Insert a class module. From the menu bar, click Insert \Rightarrow Class Module and accept the default name of Class1. Your cursor is blinking in the Class1 module's Code window.
11. The purpose of this particular class module is to capture an event that is associated with OptionButton controls. At the top of the Class1 module, publicly declare a variable that refers to the group of OptionButtons you will involve in the class module code. In that same statement, expose the events associated with OptionButtons using the WithEvents keyword. The following statement accomplishes this task:

```
Public WithEvents OptGroup As msforms.OptionButton
```

NOTE There is nothing special about the OptGroup variable name; you can give your class module variable whatever name makes sense to you. What makes sense to me is that I am grouping some OptionButton controls for a demonstration, so OptGroup is an intuitive name.

12. To demonstrate the point of this lesson, you can use the Click event for your OptGroup class. A message box displays the name and caption of the OptionButton that was clicked if that OptionButton is included in the class. Figure 24-12 shows how the VBE looks after inputting the following class module code.

```

Private Sub OptGroup_Click()
    MsgBox "Hello, my name is " & OptGroup.Name & "." & vbCrLf &
    "My caption is " & OptGroup.Caption & ".", vbInformation, _
    "You just clicked me, here is my info:"
    End Sub

```

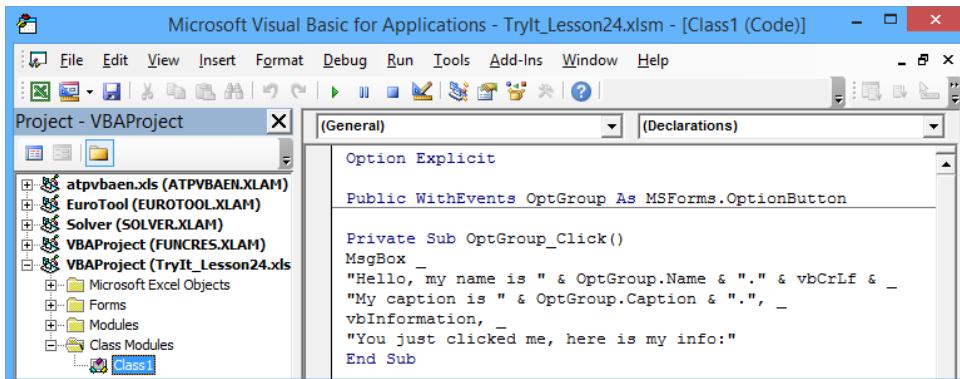


FIGURE 24-12

NOTE If this were an actual workbook project, you would not need a message box to tell you which OptionButton was just clicked. More realistically, you might assign a String type variable to the selected OptGroup.Caption if that caption string is needed as part of an operation elsewhere in your project.

13. Return to the UserForm module. At the top of the module, identify which OptionButtons you want to be grouped into the OptGroup class. For this example, the first five OptionButtons are grouped, so create an instance of the OptGroup class with the New keyword for the Class1 module name:

```
Dim optButtons(1 To 5) As New Class1
```

14. The UserForm's Initialize event is a good opportunity to do the actual grouping of the five OptionButtons. From the Object drop-down list select UserForm, and in the Procedure drop-down list select Initialize. VBA enters the UserForm_Initialize and End Sub statements with an empty space between the two lines, as follows:

```
Private Sub UserForm_Initialize()
End Sub
```

15. Declare an Integer type variable that helps loop through the five OptionButtons that become a part of the class module:

```
Dim intCounterOptionButton As Integer
```

16. Open a For Next loop to loop through the five OptionButtons:

```
For intCounterOptionButton = 1 To 5
```

17. Set each of the five OptionButtons as members of the OptGroup class:

```
Set optButtons(intCounterOptionButton).OptGroup = _
Controls("OptionButton" & intCounterOptionButton)
```

18. Continue and close the For Next loop with the Next statement:

```
Next intCounterOptionButton
```

19. All of your coding is complete. The entire UserForm module contains the following VBA code:

```
Option Explicit

Dim optButtons(1 To 5) As New Class1
Private Sub UserForm_Initialize()
Dim intCounterOptionButton As Integer
For intCounterOptionButton = 1 To 5
Set optButtons(intCounterOptionButton).OptGroup = _
Controls("OptionButton" & intCounterOptionButton)
Next intCounterOptionButton
End Sub

Private Sub cmdExit_Click()
Unload Me
End Sub
```

20. Test your class module by showing the UserForm. Press Ctrl+G to open the Immediate window, type the statement `UserForm1.Show`, and then press Enter.
21. Click any of the five OptionButtons on the left to display the message box that identifies the name and caption of the OptionButton you click. In Figure 24-13 I clicked OptionButton4, which has the caption Grapes. The OptionButtons on the right side of the UserForm are not included in the class, and if clicked do not invoke a message box.

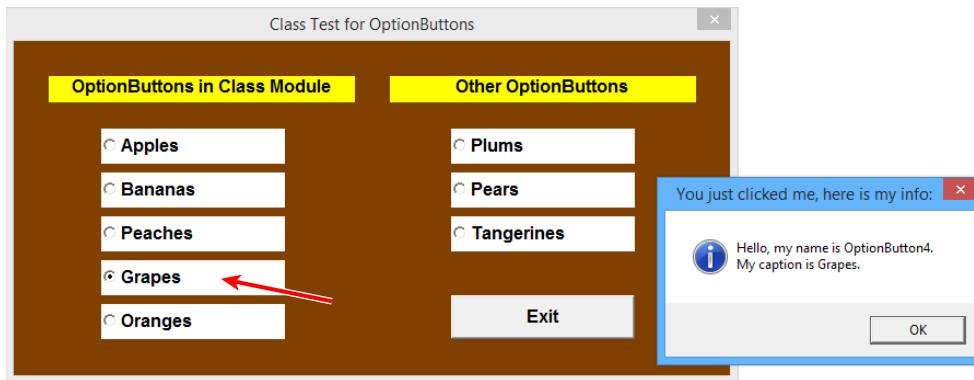


FIGURE 24-13

REFERENCE Please select the video for Lesson 24 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

25

Add-Ins

Add-ins are a useful feature in Excel, considered by many Excel developers to be an indispensable tool when distributing their custom projects to a wider audience. Anyone can create an add-in—it's the kind of thing that's easy to do once you know how. This lesson discusses the concept of add-ins and how to incorporate them into your Excel projects.

NOTE *This lesson discusses standard Excel add-ins. Two other types of add-ins exist that are not developed with VBA and are not discussed in this lesson. One of the other types is called COM add-ins, developed with languages such as Visual Basic, C++, and J++ that support component object model components. The other type is DLL add-ins, which are Windows files known as Dynamic Link Library files.*

WHAT IS AN EXCEL ADD-IN?

An Excel add-in is a special type of Excel workbook that has been converted to an add-in file. There is no magic to the add-in conversion process, but after you create an add-in file, you'll notice its unique characteristics:

- The file extension is .xla for Excel versions prior to 2007, and .xlam for Excel versions 2007 through 2013.
- Add-ins are always hidden; you do not open and view them as you would an Excel workbook.
- You cannot show sheets of any kind belonging to the add-in file.
- The add-in file is not recognized as an open workbook in the `Workbooks` collection.

WHY CREATE AN EXCEL ADD-IN?

Add-ins commonly use VBA macros, event procedures, user-defined functions, and UserForms to make everyday tasks faster and easier to accomplish. Many Excel users don't find the need to create an add-in, but here are some reasons why you might want to:

- Add-in files are hidden and therefore provide seamless integration to open Excel workbooks. Novice Excel users don't need to worry about opening an add-in after it's been loaded, and they won't wonder about an extra open Excel file because add-ins cannot be seen or unhidden.
- Even if the macro security is set to its most restrictive level, the VBA programming for an installed add-in can still run.
- Add-ins open automatically when Excel starts.
- The custom feature(s) contained within the add-in file are usually available to any of the open workbooks.
- The programming code is contained in the add-in file itself, and does not travel with the workbooks that use it. This gives you more control over how the file is distributed and who can access its code.
- Add-ins really shine in their ability to perform actions on several objects, such as cells or sheets, that if done manually would be cumbersome, time-consuming, and require some knowledge of Excel for the user to complete. Novice Excel users will especially appreciate the ease of clicking a button to do tasks that they might not know how to do manually, or might not know the most efficient methods by which to handle those tasks quickly.

CREATING AN ADD-IN

You create an Excel add-in file manually, but you make its features available by using VBA. To create an add-in, the first thing you do is open a new workbook. Because you add VBA code that becomes the add-in's functionality, you should test and retest your code before releasing the add-in for others to use. I mention this obvious point because if your add-in deals with manipulating worksheets in the active file, you need to observe the code's effect on those worksheets to make sure everything is working properly. After you convert the workbook to an add-in, you're no longer able to view the worksheets, so you want to construct and test all your code before converting your workbook as an add-in.

Suppose you want to create an add-in that offers the options to hide, unhide, protect, or unprotect multiple worksheets. A novice Excel user might perform these tasks one sheet at a time—quite an undertaking if the workbook contains dozens or hundreds of worksheets and the tasks are a frequent chore.

PLAN AHEAD FOR BEST RESULTS

You can convert any workbook to an add-in file, but not every workbook is a good candidate as an add-in. When I create an add-in, I know in advance what features I want the add-in to have, and what kind of code to avoid. This is important, because the add-in file is a hidden workbook that cannot contain code for activating a sheet or a range of cells.

You can write data to your add-in file, but you cannot activate the add-in file at any time. If you want to keep any data you've written to the add-in file, you need to save the file in the `Workbook_BeforeClose` event, because when an add-in closes, it does not prompt the user to save unsaved changes.

In your new workbook that is destined to become an add-in, press Alt+F11 to go to the Visual Basic Editor. From the VBE menu bar, click Insert \Rightarrow UserForm. If the Properties window is not visible, press the F4 key. Follow these steps to create the add-in:

1. Select your new UserForm in its design area. In the Properties window, name the UserForm `frmSheetManager`, enter its caption as `Sheet Manager`, and set its `Height` property to `210` and its `Width` property to `276`.
2. Place the following controls on your UserForm:
 - A Label control near the top, setting its `Width` property to `228` and its `Caption` property to `Please select your action::`.
 - An OptionButton control below the Label control, keeping the default name `OptionButton1`, setting its `BackColor` property to `white`, its `Width` property to `228`, and its `Caption` property to `Unhide all sheets`.
 - A second OptionButton control below `OptionButton1`, keeping the default name `OptionButton2`, setting its `BackColor` property to `white`, its `Width` property to `228`, and its `Caption` property to `Hide all sheets except active sheet`.
 - A third OptionButton control below `OptionButton2`, keeping the default name `OptionButton3`, setting its `BackColor` property to `white`, its `Width` property to `228`, and its `Caption` property to `Protect all sheets`.
 - A fourth OptionButton control below `OptionButton3`, keeping the default name `OptionButton4`, setting its `BackColor` property to `white`, its `Width` property to `228`, and its `Caption` property to `Unprotect all sheets`.
 - A CommandButton near the bottom-left corner of the UserForm, setting its `Name` property to `cmdOK`, and its `Caption` property to `OK`.
 - A CommandButton near the bottom-right corner of the UserForm, setting its `Name` property to `cmdExit`, and its `Caption` property to `Exit`.

Your UserForm ends up looking like Figure 25-1.

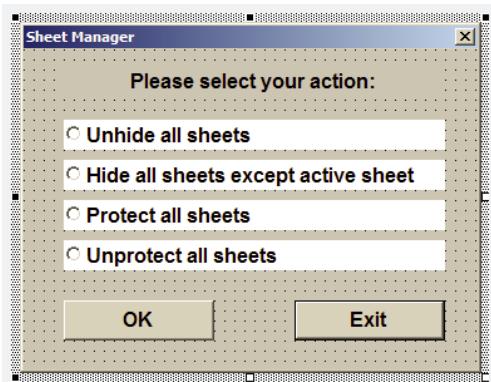


FIGURE 25-1

The design work is complete for your UserForm. In the UserForm module, enter the following code, which is mostly triggered by the cmdOK button's Click event. The requested task is performed depending on which OptionButton was selected:

```

Private Sub cmdOK_Click()
    'Declare an Integer type variable to help loop through the worksheets.
    Dim intSheet As Integer

    'Open a Select Case structure to evaluate each OptionButton.
    Select Case True

        'If OptionButton1 was selected to unhide all sheets:
        Case OptionButton1.Value = True
        For intSheet = 1 To Sheets.Count
            Sheets(intSheet).Visible = xlSheetVisible
        Next intSheet

        'If OptionButton2 was selected to hide all sheets except active sheet:
        Case OptionButton2.Value = True
        For intSheet = 1 To Sheets.Count
            If Sheets(intSheet).Name <> ActiveSheet.Name Then
                Sheets(intSheet).Visible = xlSheetHidden
            End If
        Next intSheet

        'If OptionButton3 was selected to protect all sheets.
        Case OptionButton3.Value = True
        For intSheet = 1 To Sheets.Count
            Sheets(intSheet).Protect
        Next intSheet

        'If OptionButton4 was selected to unprotect all sheets.
        Case OptionButton4.Value = True
        For intSheet = 1 To Sheets.Count
            Sheets(intSheet).Unprotect
        Next intSheet
    End Select
End Sub

```

```

'If no OptionButton was selected:
Case Else
MsgBox "No Action option was selected", , "Please select an option"

'Close the Select Case structure.
End Select

End Sub

Private Sub cmdExit_Click()
Unload Me
End Sub

```

Create a small macro to call the UserForm. From the VBE menu bar, click Insert \Rightarrow Module and enter the following macro:

```

Private Sub SheetManager()
frmSheetmanager.Show
End Sub

```

After completing the VBA functionality that your add-in provides to its users, it's almost time to convert the workbook to an add-in. There is an additional step you can take to add a description to the file's Properties information. It's purely optional that you do this, but it's a good habit to get into because it helps the add-in's users know what the add-in does.

The process for accessing the file's Properties information depends on your version of Excel. To access the Properties dialog box in Excel versions prior to 2007, click File \Rightarrow Properties from the worksheet menu bar as shown in Figure 25-2. In the Properties dialog box, some fields may already be entered for you by default. As you see later in this lesson, the most useful information to enter is the Title and Comments fields, as indicated in Figure 25-3.

To reach the Properties information in Excel version 2007, click the round Office button near the top-left corner of your window. You see a vertical pane on the left side of the window. Click Prepare, and then in the pane on the right, click Properties, as shown in Figure 25-4.

To reach the Properties information in Excel version 2010 and 2013, click the File tab on the Ribbon, and in the vertical pane at the left, click Info. At the far right, you see a Properties label with a drop-down arrow. As indicated in Figure 25-5, selecting the Advanced Properties item in the drop-down list displays the Properties dialog box.

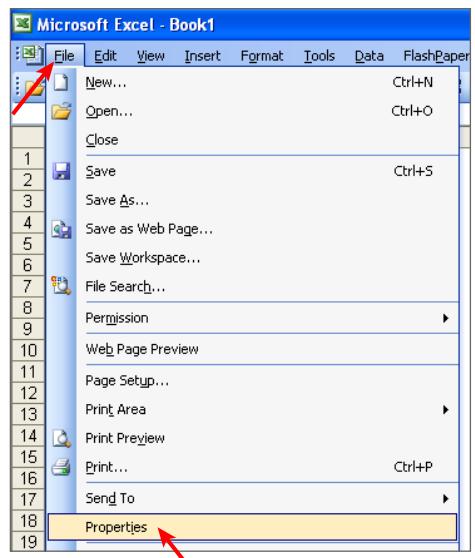


FIGURE 25-2

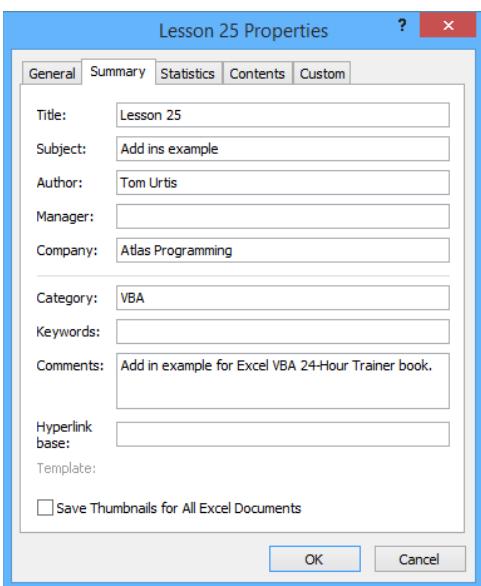


FIGURE 25-3

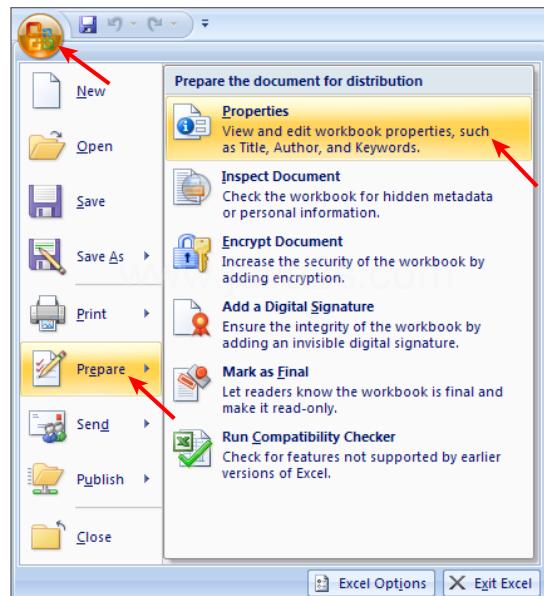


FIGURE 25-4

Lesson 25 - Excel

tom@atlaspm.com

Properties

- Show Document Panel
- Advanced Properties

Last Modified Today, 3:19 PM
Created 1/19/2011 6:43 PM
Last Printed

FIGURE 25-5

CONVERTING A FILE TO AN ADD-IN

The easiest way to convert a file to an add-in is to save the file as an Excel Add-in type. In versions of Excel prior to 2007, from the worksheet menu click File → Save As. In the Save As dialog box, navigate to the folder where you want the add-in to reside. In Figure 25-6, I named the file SheetManager, and I created a subfolder named My Addins. From the Save As Type field's drop-down list, select Microsoft Office Excel Add-In as shown in Figure 25-6, and click Save.

For version 2007, click the Office button and select Save As. For versions 2010 and 2013, click the File tab and select Save As. In the Save As dialog box, navigate to the folder where you want the add-in to reside and give the file a name. As shown in Figure 25-7, select Excel Add-In from the Save As Type drop-down list and click Save.

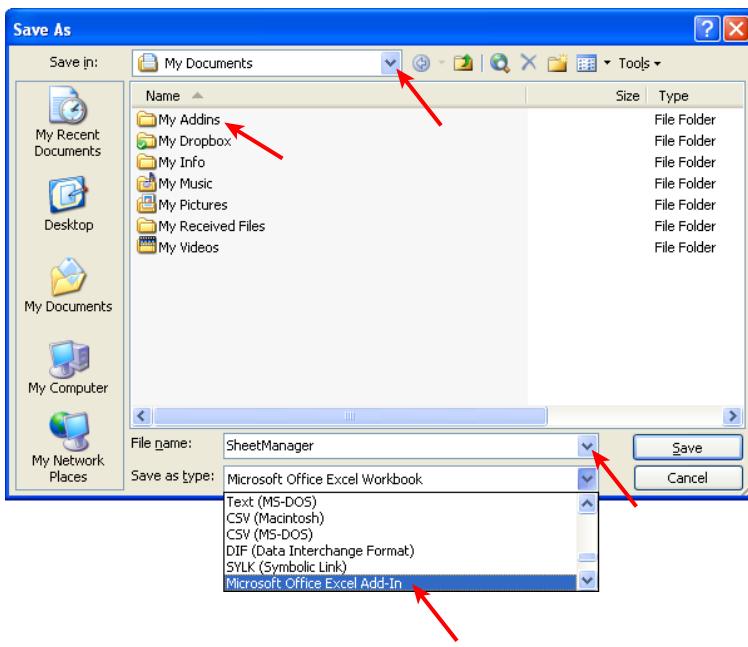


FIGURE 25-6

NOTE While saving a file as an add-in, you must have a worksheet be the active sheet. If by chance you have a chart sheet in your file and it is the active sheet, the Save As Type drop-down list won't include an Add-in file type.

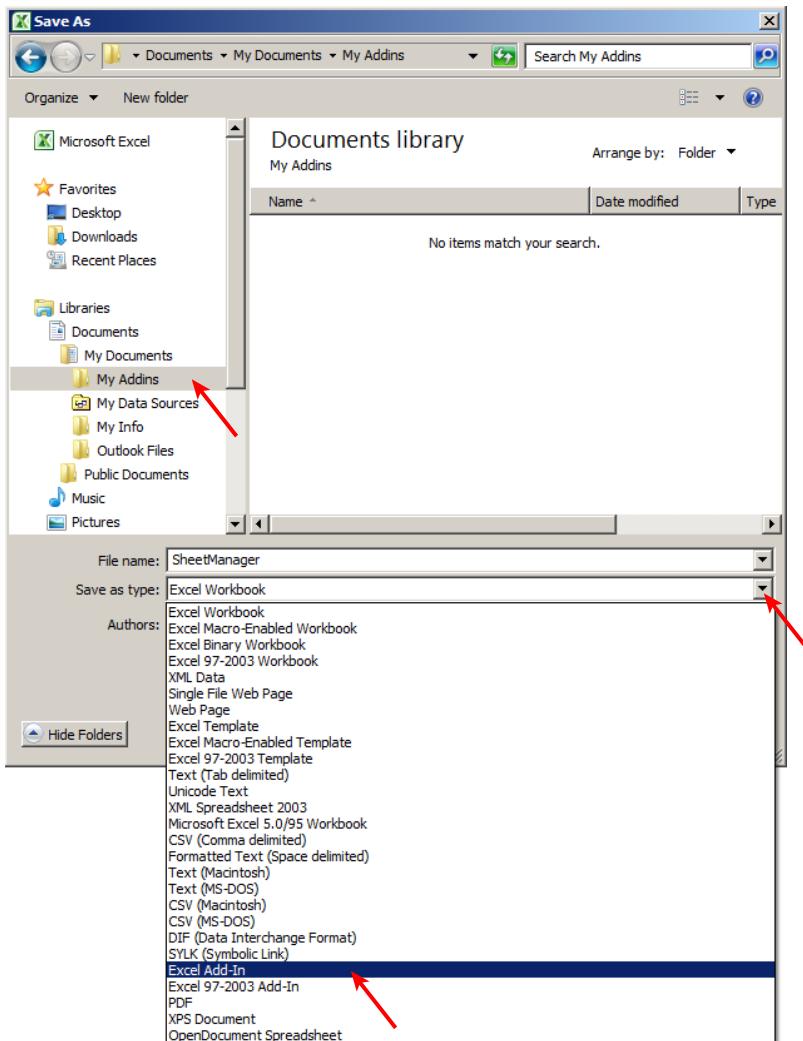


FIGURE 25-7

INSTALLING AN ADD-IN

If your add-in is being distributed to other users, the first thing you do is to deliver the add-in file to them in some way, such as by e-mail, or on a flash drive if by hand delivery. In any case, your users would save the add-in file to whatever folder they prefer, similar to how you saved your add-in file into a folder on your computer.

The easiest way to install an add-in is to use the Add-Ins dialog box, which you can do from any open workbook. In versions of Excel prior to 2007, from the worksheet menu click Tools → Add-Ins as shown in Figure 25-8. In versions 2007 to 2013, click the Developer tab on the Ribbon, and select the Add-Ins icon as shown in Figure 25-9. An example of the Add-Ins dialog box is shown in Figure 25-10.

The Add-Ins dialog box shows a list of all the add-ins that Excel is aware of. An add-in is open if a check mark is next to its name in the list. Notice in Figure 25-10 that no add-ins are selected, and that the SheetManager add-in is not listed in the Add-Ins dialog box.

When a new add-in is created, it does not automatically appear in the Add-Ins dialog box. To install a new add-in, you first need to list it in the Add-Ins dialog box, and then select it in the list.

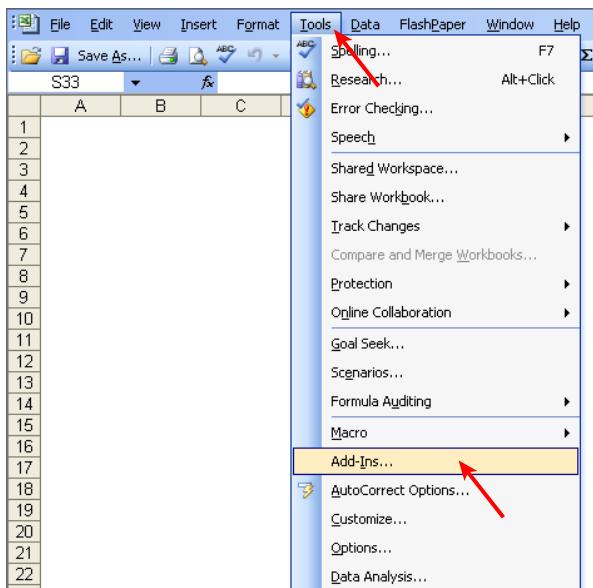


FIGURE 25-8

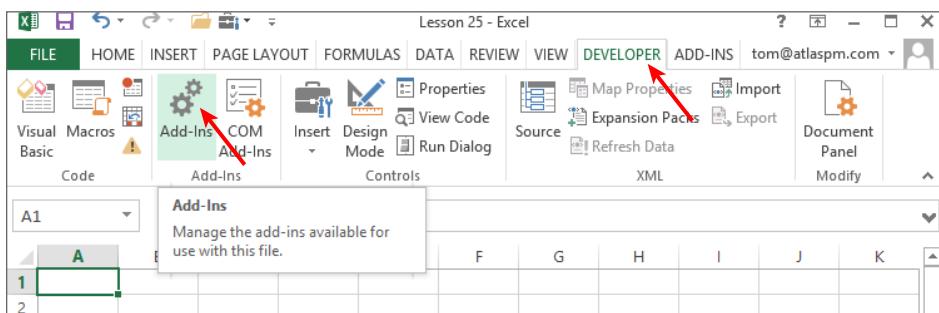


FIGURE 25-9

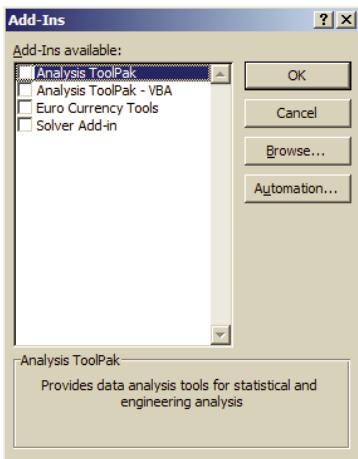


FIGURE 25-10

NOTE *The Developer tab is a very useful item to place on your Ribbon. See the section named “Accessing the VBA Environment” in Lesson 2 for the steps to display the Developer tab.*

NOTE *A quick way to open the Add-Ins dialog box from any version of Excel is to press Alt+T+I—that is, hold down the Alt key and with your other hand press the T key and the I key. If you prefer to work with a mouse instead of the keyboard, and you prefer not to show the Developer tab, you can access the Add-Ins dialog box another way. In Excel version 2007, click the Office button, then click the Excel Options button. In Excel versions 2010 and 2013, click the File tab, click the Options menu item, and select the Add-Ins menu item. At the bottom of the window, select Excel Add-Ins from the Manage drop-down list, and click Go.*

To include an add-in on the Add-Ins list, click the Browse button on the Add-Ins dialog box. Navigate to the folder where you saved the add-in file, select the filename, and click OK to exit the Browse dialog box as indicated in Figure 25-11.

You now see your selected file listed in the Add-Ins dialog box. By default, Excel places a check mark next to the selected add-in’s name. If you don’t want the add-in to be open—that is, for its features to be available to you—simply deselect the add-in by unchecking the box next to its name.

If and when you do select your new add-in, you and the users of that add-in will appreciate the extra time you spent in the Properties window before you converted the original file to an add-in. Notice that the selected add-in's filename and comments appear at the bottom of the Add-Ins dialog box, informing the user what the add-in does. In any case, now that you've listed the add-in file, click OK to exit the Add-Ins dialog box as indicated in Figure 25-12.

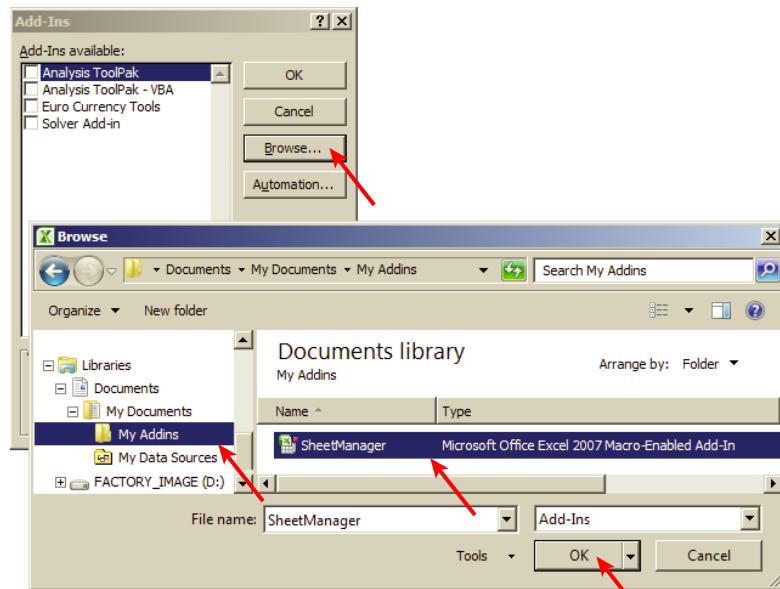


FIGURE 25-11

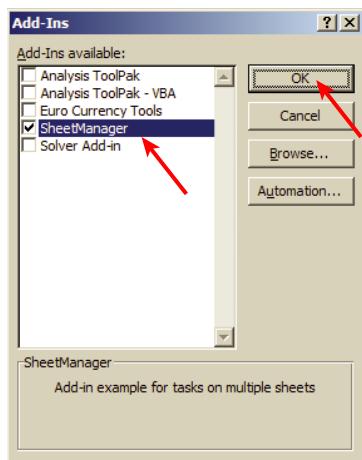


FIGURE 25-12

WHERE DID THOSE OTHER ADD-INS COME FROM?

Even before you created your first add-in, you saw some add-ins already listed in the Add-Ins dialog box. Excel ships with four available add-ins, which are not open until you select them in the Add-Ins dialog box:

- The Analysis ToolPak add-in, which provides an expanded set of analysis tools not available in standard worksheet functions and features
- The Analysis ToolPak VBA add-in, which provides an expanded set of functions for your VBA programming code
- The Euro Currency Tools add-in, which is a tool for converting and formatting the euro currency
- The Solver add-in, which is a what-if analysis tool that attempts to find an optimal value for a formula in one cell while considering constraints placed on the values in other cells

CREATING A USER INTERFACE FOR YOUR ADD-IN

Now that the add-in has been created and installed, you need to provide your users with the ability to access the functionality. As it stands right now, all that's happened is the add-in is available behind the scenes. However, because the `SheetManager` add-in's functionality is tied to a `UserForm`, you need to establish a way for users to click a link of some kind that calls the `UserForm`.

Before the Ribbon came along, a custom worksheet menu item was created using the `CommandBar` object. For this example, I named the menu item `SheetManager`, and it appears on the Tools menu. The good news is, Excel versions 2007 through 2013 still support `CommandBars`, and you can use the same code to achieve a user-friendly custom menu interface that is compatible with every version of Excel starting with Excel 97.

For versions of Excel prior to 2007, a menu item named `Sheet Manager` is in the Tools menu, as shown in Figure 25-13. For versions 2007, 2010, and 2013, the menu item named `Sheet Manager` is in the `Menu Commands` section of a new tab on the Ribbon named `Add-Ins`. The `Add-Ins` tab appears when you apply custom add-in code. In any case, clicking the `Sheet Manager` menu item executes the macro that calls the `UserForm`, as shown in Figure 25-14.

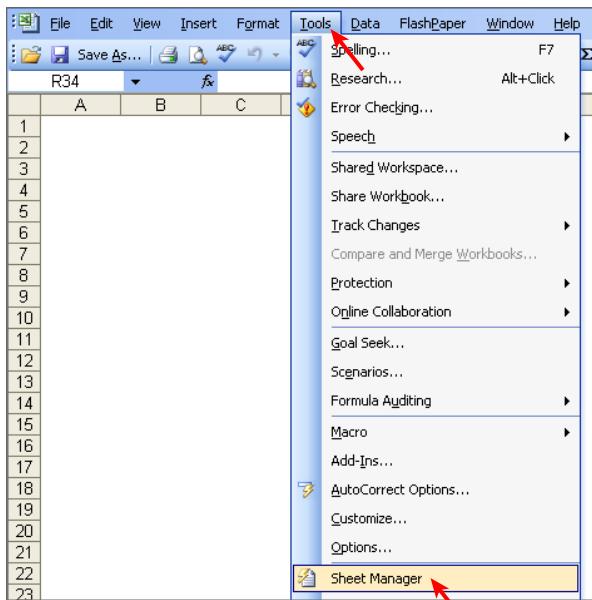


FIGURE 25-13

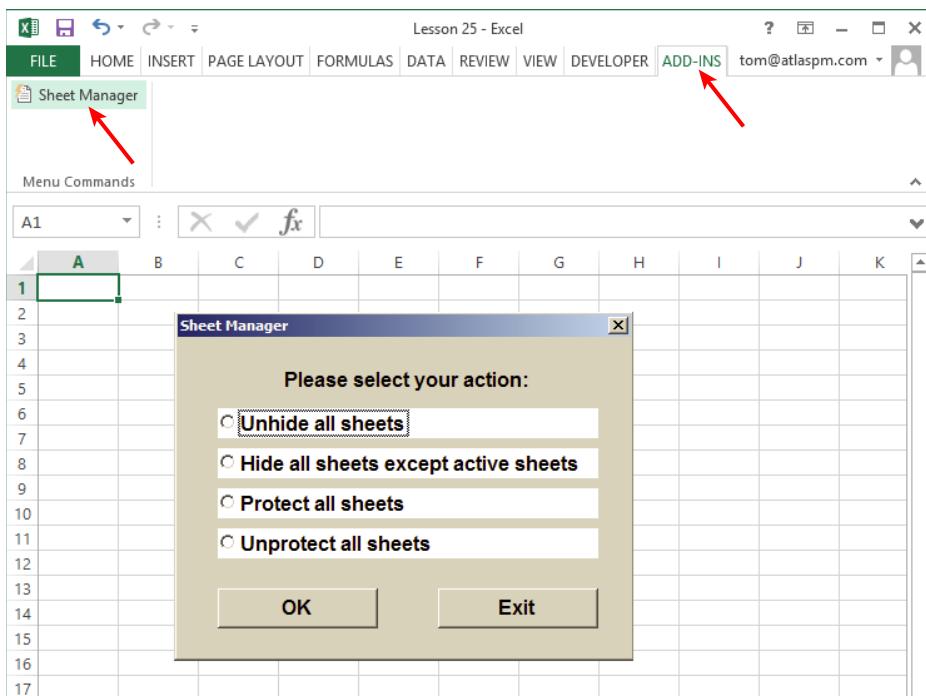


FIGURE 25-14

The following event code, found in the ThisWorkbook module of the add-in file, establishes the custom user interface:

```

Private Sub Workbook_Open()

    'Declare a CBC variable for the custom menu item.
    Dim objCmdControl As CommandBarControl
    'The custom menu item will be named "Sheet Manager"
    'and it will go onto the Tools menu for versions before 2007.
    Set objCmdControl = _
        Application.CommandBars("Worksheet Menu Bar") _
        .Controls("Tools").Controls.Add

    'For the new menu item, give it a meaningful caption,
    'help it to clearly stand out by starting a BeginGroup.
    'The OnAction method will call the UserForm.
    'The Face ID is a small icon next to the menu item
    'that is optional, but adds a feeling of customization.
    With objCmdControl
        .Caption = "Sheet Manager"
        .BeginGroup = True
        .OnAction = "SheetManager"
        .FaceId = 144
    End With

End Sub

Private Sub Workbook_BeforeClose(Cancel As Boolean)
    'Delete the custom menu item from the Tools menu.
    'The error bypass is for cases when the "Sheet Manager"
    'item is not listed on the Tools menu.
    On Error Resume Next
    Application.CommandBars("Worksheet Menu Bar") _
        .Controls("Tools").Controls("Sheet Manager").Delete
    Err.Clear
End Sub

```

Changing the Add-In's Code

You'll find that some of your add-ins are a work in progress. Users will enjoy the ease of performing add-in tasks, and you'll be requested to make enhancements to the add-in for more functionality. As you pick up more VBA programming skills, you'll want to improve your original code by making edits for speed and efficiency.

You make any changes to your add-in file in the Visual Basic Editor. Open your add-in file, and all you see is an empty-looking Excel file because all the sheets in an add-in are hidden and cannot be viewed. Press Alt+F11 to go to the VBE, and just as if it were any Excel workbook, make whatever changes to the code you need to make. When you are done, save your changes in the VBE and close the add-in file.

NOTE For add-ins that you distribute to other users, you want to protect the code from being inadvertently changed or viewed by others. The process for protecting your add-in code is the same as with any Excel workbook, and that is to lock and protect the project in the Visual Basic Editor. The steps to do this are discussed in Lesson 4, in the section “Locking and Protecting the VBE.”

CLOSING ADD-INS

As you saw in the section “Changing the Add-In’s Code,” you can open an add-in file, but you might like to know how to close an add-in file because it cannot be closed the same way you close a workbook. You have three ways to close an add-in file:

- Deselect (uncheck) the add-in’s name in the Add-Ins dialog box.
- Go into the VBE and press Ctrl+G to ensure that the Immediate window is open. In the Immediate window, enter a line of code that closes the add-in file and press Enter. An example of such code for the SheetManager add-in is as follows:

```
Workbooks ("SheetManager.xlam").Close
```
- Close Excel, which closes all files, including add-ins.

REMOVING AN ADD-IN FROM THE ADD-INS LIST

At some point in the future, you might want to remove the add-in from the list of available add-ins in the Add-Ins dialog box, if the add-in is outdated or you just don’t need it anymore. To accomplish this is an example of how science meets art, because Excel does not have a built-in way to remove an add-in’s name from the list. Here are the steps to make this happen:

1. Close Excel.
2. Open Windows Explorer and navigate to the folder that holds your add-in file.
3. Select the add-in filename, and without opening the file, either change its name, drag the file to a different folder, or, if you really no longer need the add-in, delete the file altogether.
4. Open Excel, and when you do, you receive a message telling you that the add-in file cannot be found. Click OK as indicated in Figure 25-15.

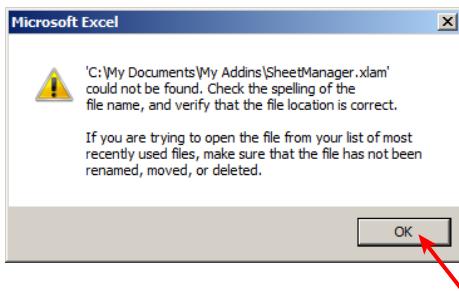


FIGURE 25-15

5. Open the Add-Ins dialog box and uncheck the name of the add-in you want to remove. Excel reminds you that the file cannot be found, and asks for confirmation that you want to delete the file from the list of available add-ins. Click Yes as indicated in Figure 25-16.

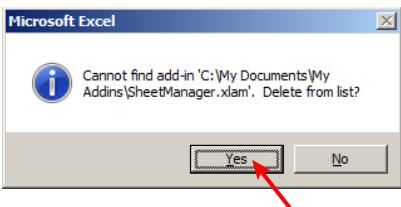


FIGURE 25-16

TRY IT

For this lesson, you create, install, and test an add-in that contains a user-defined function to return the text of another cell's comment.

Lesson Requirements

To get the sample workbook, you can download Lesson 25 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open a new workbook.
2. Go to the Properties window. In the Title field enter **Comment Text** and in the Comments field enter **Return text of comments in other cells**.
3. Exit the Properties window and press Alt+F11 to go into the Visual Basic Editor.
4. From the menu bar in the VBE, click Insert \Rightarrow Module. Copy the following user-defined function into the module:

```
Function GetComment(rng As Range) As String
Dim strText As String
If rng.Comment Is Nothing Then
    strText = "No comment"
Else
    strText = rng.Comment.Text
End If
GetComment = strText
End Function
```

5. Press Ctrl+S to display the Save As dialog box. Navigate to the folder into which you want to save this file. Name the file **CommentText** and select Excel Add-In in the Save As Type field, as indicated in Figure 25-17. Click Save, which converts this workbook as a new add-in file named **CommentText.xlam**.

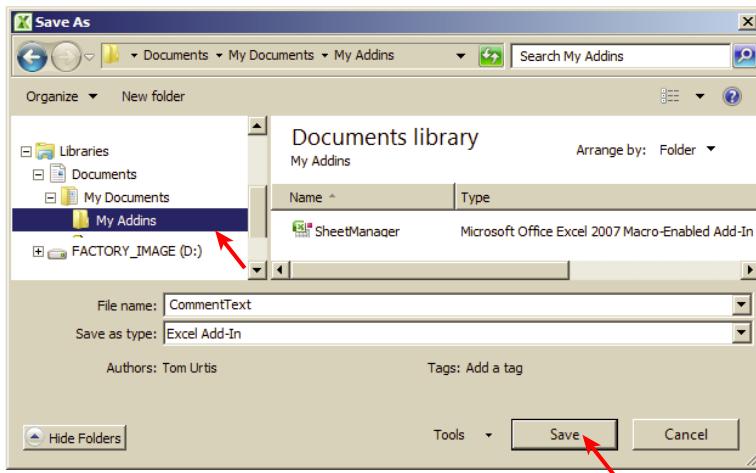


FIGURE 25-17

6. Close Excel.
7. Restart Excel and open a new workbook.
8. Right-click cell B2 of the active worksheet, and select Insert Comment. Enter some text in your comment.
9. Select cell G1.
10. Press Alt+T+I to show the Add-Ins dialog box.
11. Click Browse and navigate to the folder where you saved the CommentText add-in file. Select the CommentText file and click OK. Your Add-Ins dialog box looks like Figure 25-18, with the CommentText add-in loaded. Recall that the file is named CommentText but the Add-Ins dialog box shows it as Comment Text, and also shows the description of the add-in, because that is the information you entered in Step 2 about the add-in file in its Properties dialog box. Click OK to exit the Add-Ins dialog box.

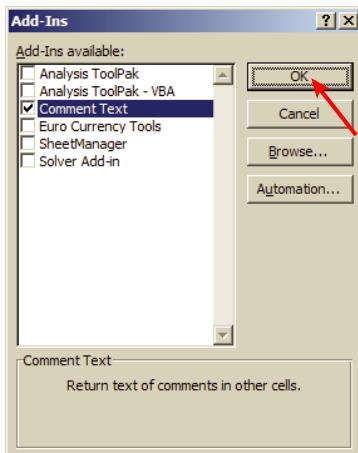


FIGURE 25-18

12. In cell G1, enter the user-defined function =GetComment(B1) and press Enter. Copy the formula down to cell G2. You see that the UDF returned No comment in cell G1 because no comment exists in cell B1. However, you did enter a comment into cell B2 in Step 8, so the UDF in cell G2 returns the text of the comment from cell B2. Your worksheet looks similar to Figure 25-19.

FIGURE 25-19

13. Note that the workbook you are looking at does not contain the GetComment UDF code. You can utilize that UDF because its code belongs to the CommentText add-in file that you installed for the active workbook.

REFERENCE Please select the video for Lesson 25 at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

26

Managing External Data

One of the most versatile and useful benefits of Excel is its ability to import data from external sources. Lessons 29–33 include examples of sharing data back and forth with other Microsoft Office applications from Excel.

Prior to Excel 97, data in an Excel workbook was entered manually. An Excel workbook was essentially a self-contained object, having almost no contact with the outside world except for the person working in the project.

Starting with Office 97, Microsoft became devoted to providing more and better tools for importing and exporting data to the Internet, database programs, and text-related software applications. Excel leads the way in this effort among all Office applications. In this lesson, you learn how to use VBA to share data between Excel and other external sources, including Access, the Internet, and text files.

CREATING QUERYTABLES FROM WEB QUERIES

The Internet as we know it has only been around since the mid-1990s—not that long ago really—but it's hard to imagine what life would be like today without the World Wide Web. The public's desire is only increasing for access to the galaxy of information that is stored on the web. With each new release of its Office suite, Microsoft has improved the capacity of its applications to interact with web-based information.

NOTE When you connect Excel to an external source such as the Internet, you add a QueryTable to your worksheet. Objects that can connect to external data sources include a cell range, an Excel table, a pivot table, a text file, and a web query. In this case, you are adding a QueryTable to a worksheet because you are querying the web for information that will be displayed on your worksheet.

Suppose you are interested in monitoring the stock prices of a half-dozen or so technology companies. If you want to avoid the monotony of going to a financial website and entering the

same stock symbols every time, you can automate the process with a web query, and refresh the data anytime you like.

When you build a web query, you need to tell Excel the website from which to extract the information, and the cell address on the destination sheet where you want the QueryTable to be located. Some background information about URLs and their parameters might be helpful for you to understand what is going on.

If you open your web browser and enter the URL `http://money.cnn.com/quote/quote.html?symb=YHOO+GOOG`, you reach a site that provides a table of stock quotes for Yahoo! and Google. With this URL, you are essentially passing URL parameters that enable you to pass information such as search criteria to a website. In this case, the URL parameters being used are the symbols for Yahoo! (YHOO) and Google (GOOG).

The following macro places the QueryTable on cell A1, and points to one of the bevy of websites out there that provide current stock quotes. For demonstration purposes, I chose a few companies that are all headquartered in the Silicon Valley area where I live and own my Excel development company. The stock symbols of those companies are the criteria that apply URL parameters through the code to gather the stock quote information that populates the QueryTable. Figure 26-1 shows what the result looked like when I ran this macro in November 2014:

```
Sub ImportStocks()

'Declare variables for destination worksheet,
'and two halves of the connection string:
'one half for the URL, and the other half for
'the quotes, to make it easier for you to edit.
Dim wsDestination As Worksheet
Dim strURL As String, strStocks As String

'Set your preferred destination worksheet; here it is Sheet2.
Set wsDestination = Worksheets("Sheet2")

'Define the URL for getting your stock quotes.
'There are many websites where you can do this.
strURL = "http://money.cnn.com/quote/quote.html?symb="

'Define your stocks of interest. I only selected these
'as an example of nearby Silicon Valley businesses.
strStocks = "AAPL,CSCO,EBAY,GOOG,INTC,ORCL,YHOO"

'My preference is to activate the destination worksheet
'and select cell A1.
Application.Goto wsDestination.Range("A1"), True

'Clear the cells in the worksheet so you know the data
'being imported will not be confused with other data
'you may have imported previously and not yet deleted.
Cells.Clear

'Add your QueryTable with the connection string
'and other useful methods you see in the With structure.
With wsDestination.QueryTables.Add _
(Connection:="URL;" & strURL & strStocks, _
Destination:=Range("$A$1"))
```

```

    .BackgroundQuery = True
    .SaveData = True
    .AdjustColumnWidth = True
    .WebSelectionType = xlSpecifiedTables
    .WebFormatting = xlWebFormattingNone
    .WebTables = """wsod_multiquoteTable"""
    .Refresh BackgroundQuery:=False
End With

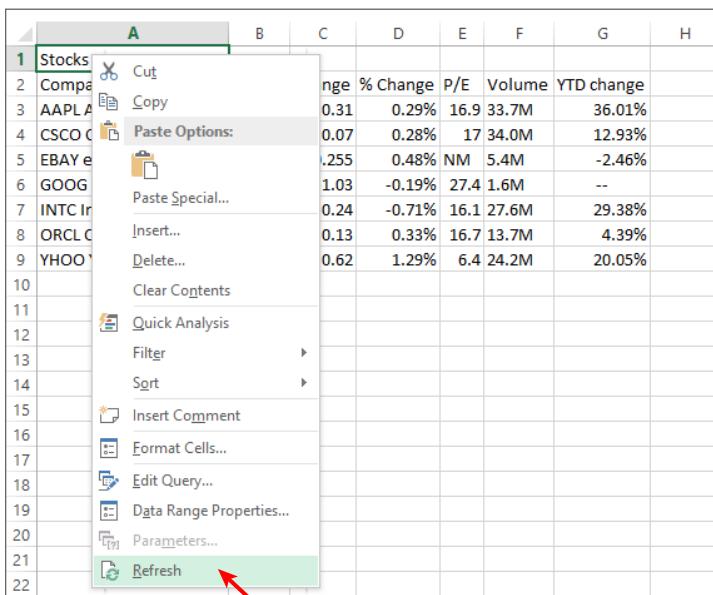
'Release object variable memory.
Set wsDestination = Nothing
End Sub

```

	A	B	C	D	E	F	G	H
1	Stocks							
2	Company	Price	Change	% Change	P/E	Volume	YTD change	
3	AAPL Apple Inc	109.01	0.31	0.29%	16.9	33.7M	36.01%	
4	CSCO Cisco Systems Inc	25.33	0.07	0.28%	17	34.0M	12.93%	
5	EBAY eBay Inc	53.52	0.255	0.48%	NM	5.4M	-2.46%	
6	GOOG Google Inc	541.01	-1.03	-0.19%	27.4	1.6M	--	
7	INTC Intel Corp	33.58	-0.24	-0.71%	16.1	27.6M	29.38%	
8	ORCL Oracle Corp	39.94	0.13	0.33%	16.7	13.7M	4.39%	
9	YHOO Yahoo! Inc	48.55	0.62	1.29%	6.4	24.2M	20.05%	
10								

FIGURE 26-1

With the worksheet active, you can refresh the data by right-clicking cell A1 and selecting Refresh, as shown in Figure 26-2. Alternatively, you can execute the VBA expression Range("A1").QueryTable.Refresh in the Immediate window or in a macro. Each time you refresh the data, you see the most recent version of the information in the data source, including any changes that were made to the data.

**FIGURE 26-2**

NOTE Does your web query take too long to refresh? You can cancel the Refresh method if it's running longer than you want to wait with this block of code:

```
If Application.Wait(Now + TimeValue("0:00:10")) Then
    With Worksheets(1).QueryTables(1)
        If .Refreshing Then
            .CancelRefresh
            MsgBox "Refresh was cancelled.", , "FYI..."
        End If
    End With
End If
```

While on the subject of corporate performance, the following macro opens a .csv file for you, depending on which stock symbol you are searching for, and copies several years of historical stock price activity to Sheet3 of your workbook:

```
Sub ImportHistory()
    Dim strStockSymbol As String
    Dim strURL1 As String, strURL2 As String

    'Download the past years' stock price activity.
    strURL1 = "http://ichart.finance.yahoo.com/table.csv?s="
    strURL2 = "&d=2&e=18&f=2010&g=d&a=2&b=13&c=1986&ignore=.csv"
    strStockSymbol = "EBAY"

    Workbooks.Open Filename:=strURL1 & strStockSymbol & strURL2

    'Copy data from the csv file to your worksheet.
    Range("A1").CurrentRegion.Copy _
    ThisWorkbook.Worksheets("Sheet3").Range("A1")

    'Close the csv file without saving it.
    ActiveWorkbook.Close False

    'Autofit the columns.
    Columns.AutoFit
End Sub
```

NOTE Another example in the Try It section leads you in a step-by-step process of creating a web query.

CREATING A QUERYTABLE FOR ACCESS

In upcoming lessons you learn about importing and exporting data between Excel and Access, using VBA and a technology called *Structured Query Language*, or SQL. Because this lesson deals with external data, you might be interested to know how to quickly, albeit manually, import an Access table directly to your worksheet.

Click the Data tab on the Ribbon, and find the Get External Data section at the far left. Click the leftmost icon that is labeled From Access as shown in Figure 26-3.

You see the Select Data Source dialog box. Navigate to the folder holding your Access database, select the folder, and also select the name of the database file. Click Open as shown in Figure 26-4.

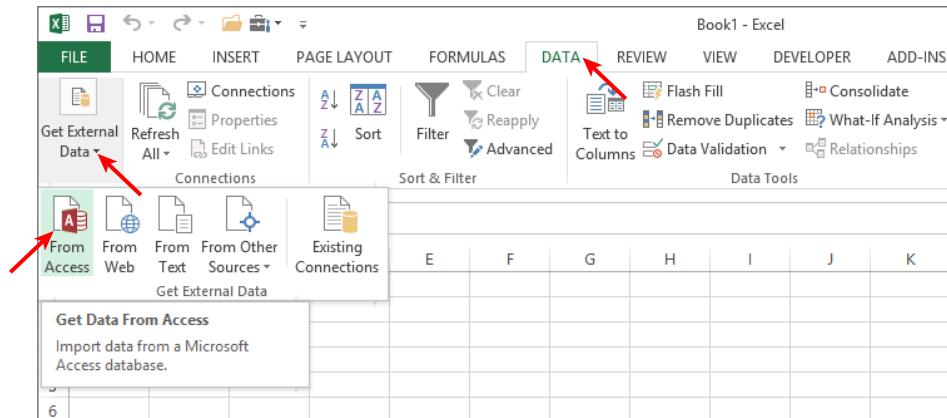


FIGURE 26-3

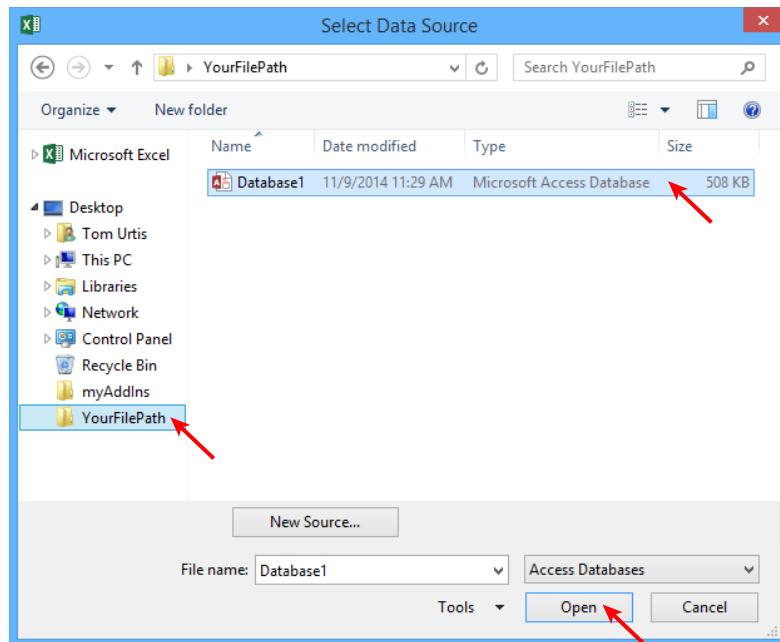


FIGURE 26-4

The Select Table dialog box displays, so all you need to do is click to select the name of the table, and then click OK as shown in Figure 26-5. After that, the Import dialog box displays. I chose to keep the imported table as a Table format, placed onto my worksheet, starting in cell A1 as shown

in Figure 26-6. Your Access table loads onto your worksheet as shown in Figure 26-7, with the top row having AutoFilter buttons to help you with your future searches.

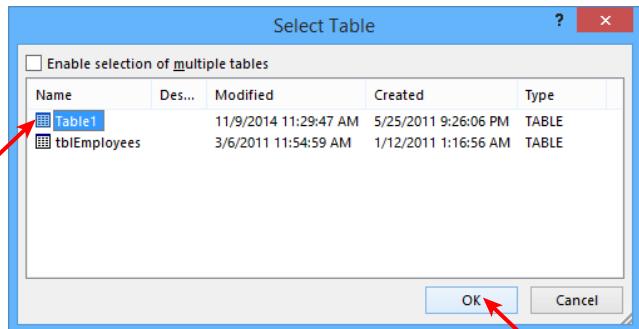


FIGURE 26-5

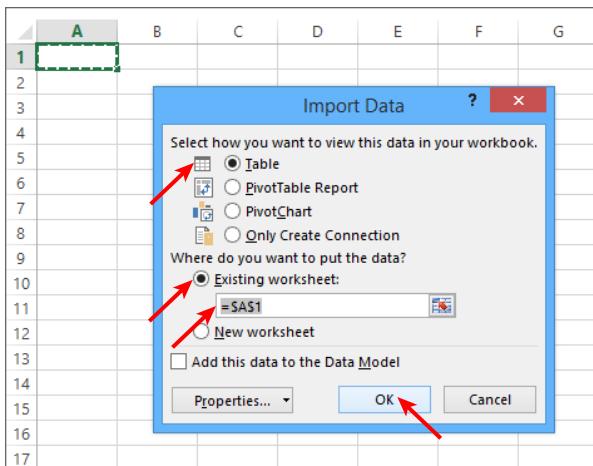


FIGURE 26-6

ID	FirstName	LastName	Gender	Title	Region	YearHired
1	Dorah	Jarre	Female	Engineer	North	2001
2	Dee	Lited	Female	Sales	South	2012
3	Anne	Teak	Female	Bookkeeper	West	2010
4	Rudy	Mentri	Male	Analyst	North	1998
5	Adam	Zapple	Male	Chef	East	2004
6	Frieda	Innosint	Female	Counsel	West	2009
7	Holden	Yermoney	Male	Accountant	South	2002
8	Faye	Kinnett	Female	Nurse	East	2013
9	Jim	Nastik	Male	Health Trainer	West	2000
10	Barb	Dwyer	Female	Security	North	2001
11	Mike	Raffone	Male	Spokesman	West	2014
12	Lance	Lyde	Male	Actuary	East	2011
13	Seymour	Klearly	Male	Optometrist	North	1996
14	Ella	Vader	Female	Maintenance	South	1999
15	Anna	Graham	Female	Communications	East	2008
16	Raynor	Schein	Male	Forecaster	East	2013

FIGURE 26-7

NOTE The Select Table dialog box may contain tables and queries, and you can import data from either of them. You might want to be aware that parameter queries do not appear in this dialog box.

USING TEXT FILES TO STORE EXTERNAL DATA

Hail the text file, the true foot soldier interface for transferring information between two or more otherwise disparate platforms. In the modern age of computing, it's always been the text file that could be relied on for one application downloading its information in a comma-delimited or fixed-length file, and another application like Excel being able to accept the data.

Text files are not pretty, they are almost never formatted, and they are not easy to read. But when all else fails, they come through and are fairly easy to program. The following examples show how text files can help you in your everyday work.

Suppose you want Excel to add a new record to a text file that records the date and time a particular Excel workbook was saved. Let's say your C drive has a folder named `YourFilePath`, which holds a text file named `LogFile.txt`. The following VBA code goes into the `ThisWorkbook` module of the Excel file you are monitoring. Modify the macro as needed for your folder path and/or name of your text file.

```
Private Sub Workbook_BeforeSave(ByVal SaveAsUI As Boolean, Cancel As Boolean)
Dim intCounter As Integer, myFileName As String
myFileName = "C:\YourFilePath\LogFile.txt"
intCounter = FreeFile
Open myFileName For Append As #intCounter
Write #intCounter, ThisWorkbook.FullName, Now, Application.UserName
Close #intCounter
End Sub
```

This macro creates four new text files, naming each with the prefix `MyFile`, followed by a number suffix in order from 1 to 4. For example, the first file is named `MyFile001.txt`, the second file is named `MyFile002.txt`, and so on. The starting number of 1 is derived by the code line `For intCounter = 1 To 4`. If you wanted to create four new text files starting with the name `MyFile038.txt`, you'd establish the starting number of 38 by specifying it with the line of code `For intCounter = 38 To 41`.

```
Sub CreateTextFiles()
Dim intCounter As Integer, strFile As String
For intCounter = 1 To 4
strFile = "MyFile" & Format(intCounter, "000")
strFile = "C:\YourFilePath\" & strFile & ".txt"
Open strFile For Output As #1
Close
Next intCounter
End Sub
```

The following macro copies the text of your comments in your worksheet's used range into a text file, where they are listed along with the cell values in that range. This is a very fast macro.

```
Sub Comment2Text()
Dim cmt As Comment, rng As Range
Dim iRow As Long, iCol As Long
Dim strText As String
Set rng = Range("A1").CurrentRegion
Open "C:\YourFilePath\YourFileName.txt" For Output As #1
For iRow = 1 To rng.Rows.Count
For iCol = 1 To rng.Columns.Count
If Not Cells(iRow, iCol).Comment Is Nothing Then
strText = strText & Cells(iRow, iCol).Text & _
"(" & Cells(iRow, iCol).Comment.Text & ")" & ";"
Else
strText = strText & Cells(iRow, iCol).Text & ";"
End If
Next iCol
strText = Left(strText, Len(strText) - 1)
Print #1, strText
strText = ""
Next iRow
Close
End Sub
```

If you want to know how many lines a particular text file has, the following macro tells you:

```
Sub Test1()
Dim MyObject As Object, LineCount As Variant
Set MyObject = _
CreateObject("Scripting.FileSystemObject")
With MyObject.OpenTextFile("C:\YourFilePath\YourFileName.txt", 1)
LineCount = Split(.ReadAll, vbCrLf)
End With
MsgBox UBound(LineCount) - LBound(LineCount) + 1
End Sub
```

Export each sheet in this workbook as a text file, with each file named as the sheet tab name. Text file macros compile very quickly.

```
Sub TextExport()
Dim rng As Range
Dim iWks As Integer, LRow As Long, iCol As Long
Dim sTxt As String, sPath As String
sPath = "C:\YourFilePath\"

For iWks = 1 To Worksheets.Count
Open sPath & Worksheets(iWks).Name & ".txt" For Output As #1
Set rng = Worksheets(iWks).Range("A1").CurrentRegion
For LRow = 1 To rng.Rows.Count
For iCol = 1 To rng.Columns.Count
sTxt = sTxt & Worksheets(iWks).Cells(LRow, iCol).Value & vbTab
Next iCol
Print #1, Left(sTxt, Len(sTxt) - 1)
sTxt = ""
Next LRow
Close #1
End Sub
```

```

Next iWks
MsgBox "The text files can be found in " & Left(sPath, Len(sPath) - 1)
End Sub

```

If you would like to see a text file's contents in a message box, you can use the following code:

```

Sub GetTextMessage()
Dim sTxt As String, sText As String, sPath As String
sPath = "C:\YourFilePath\YourFileName.txt"
If Dir(sPath) = "" Then
    MsgBox "File was not found."
    Exit Sub
End If
Close
Open sPath For Input As #1
Do Until EOF(1)
    Line Input #1, sTxt
    sText = sText & sTxt & vbCrLf
Loop
Close
sText = Left(sText, Len(sText) - 1)
MsgBox sText
End Sub

```

Suppose you want to save the contents of cell A1 on Sheet1 as a text file. The following example shows how you can do that:

```

Sub SaveCellValue()
Open "C:\YourFilePath\YourFileName.txt" For Append As #1
Print #1, Sheets("Sheet1").Range("A1").Value
Close #1
End Sub

```

Finally, this macro demonstrates how to delete a text file if it exists, and replaces it with a new text file of the same name. If the text file does not exist, the macro creates a new text file:

```

Sub DeleteAndCreate()
Dim strFile As String, intFactor As Integer
On Error Resume Next
strFile = "C:\YourFilePath\YourFileName.txt"
Kill strFile
Err.Clear
intFactor = FreeFile
Open strFile For Output Access Write As #intFactor
Close #intFactor
End Sub

```

TRY IT

What is today's date, and what is the current time of day? In this lesson you create a web query to access the website of the United States Naval Observatory, where the day and time are recorded on the Master Clock of the United States Navy. The web query imports a display of the current day and time for several North American time zones.

Lesson Requirements

To get the sample workbook, you can download Lesson 26 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. Open a new workbook.
2. From your worksheet, press Alt+F11 to go to the Visual Basic Editor.
3. From the menu bar in the VBE, click Insert \Rightarrow Module.
4. In your new module, type `Sub TimeAfterTime` and press Enter. VBA produces the following two lines of code, separated by an empty line:

```
Sub TimeAfterTime()
```

```
End Sub
```

5. Open a `With` structure for the destination worksheet:

```
With Worksheets("Sheet1")
```

6. Declare a `String` type variable for the website address:

```
Dim strURL As String
```

7. Define the website address from which the information will be imported to your worksheet:

```
strURL =  
"http://tycho.usno.navy.mil/cgi-bin/timer.pl"
```

8. For consistency, I prefer to activate the worksheet that will receive the web data. Cell A1 is a convenient cell to start with:

```
Application.Goto .Range("A1"), True
```

9. Clear the cells in the worksheet so you know the data being imported will not be confused with other data you may have imported previously and not yet deleted:

```
Cells.Clear
```

10. Open a `With` structure for the `Add` method of your new `QueryTable`. You must specify the connection, URL, destination sheet, and other information that follows:

```
With .QueryTables.Add  
(Connection:="URL;" &strURL, Destination:=.Range("A1"))  
.BackgroundQuery = True  
.TablesOnlyFromHTML = False  
.Refresh BackgroundQuery:=False  
.SaveData = True
```

11. Close the `With` structure of the `QueryTable`'s `Add` method:

```
End With
```

12. Close the With structure for the destination worksheet:

```
End With
```

13. Your entire macro looks as follows:

```
Sub TimeAfterTime()

    'Open a With structure for the destination worksheet.
    With Worksheets("Sheet1")

        'Declare a String type variable for the website address.
        Dim strURL As String
        'Define the website address, from which the information
        'will be imported to your worksheet.
        strURL =
            "http://tycho.usno.navy.mil/cgi-bin/timer.pl"

        'For consistency, I prefer to activate the worksheet
        'that will receive the web data.
        'Cell A1 is a convenient cell to situate yourself.
        Application.Goto .Range("A1"), True

        'Clear the cells in the worksheet so you know the data
        'being imported will not be confused with other data
        'you may have imported previously and not yet deleted.
        Cells.Clear

        'Open a With structure for the Add method of your new
        'QueryTable. The connection, URL, and destination sheet,
        'and other information that follows, must be specified.
        With .QueryTables.Add _
            (Connection:="URL;" &strURL, Destination:=.Range("A1"))
            .BackgroundQuery = True
            .TablesOnlyFromHTML = False
            .Refresh BackgroundQuery:=False
            .SaveData = True

        'Close the With structure of the QueryTable's Add method.
        End With

        'Close the With structure for the destination worksheet.
        End With

    End Sub
```

14. Press Alt+Q to return to the worksheet.

15. You can test the macro by pressing Alt+F8 to display the Macro dialog box as shown in Figure 26-8. Run the macro named TimeAfterTime. The result resembles Figure 26-9.

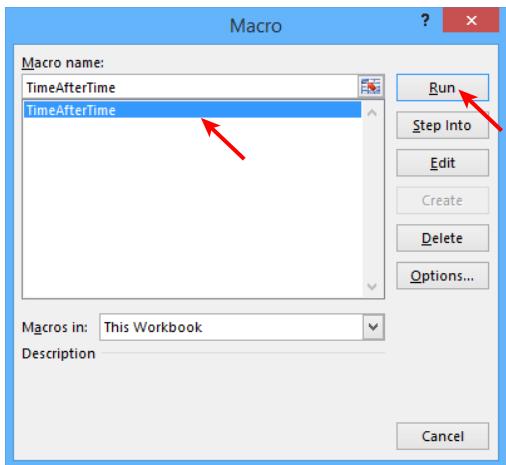


FIGURE 26-8

	A	B	C	D	E	F	G
1	US Naval Observatory Master Clock Time						
2							
3							
4	Nov. 09, 19:45:02 UTC						Universal Time
5							
6	Nov. 09, 02:45:02 PM EST						Eastern Time
7							
8	Nov. 09, 01:45:02 PM CST						Central Time
9							
10	Nov. 09, 12:45:02 PM MST						Mountain Time
11							
12	Nov. 09, 11:45:02 AM PST						Pacific Time
13							
14	Nov. 09, 10:45:02 AM AKST						Alaska Time
15							
16	Nov. 09, 09:45:02 AM HAST						Hawaii-Aleutian Time
17							
18	US Naval Observatory						
19							

FIGURE 26-9

REFERENCE Please select the video for Lesson 26 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

27

Data Access with ActiveX Data Objects

The topic of data access has become one of the most intensive forces in driving the recent development of commercial software applications. Data storage and search engine companies have become the face of the worldwide voracious demand for accessing information.

Excel is without peer in its powerful features for calculating and analyzing data, and in its ability to produce customized reports in an instant with VBA. For users who deal with extremely large volumes of source data, Excel can still fall short as a data storage application. Microsoft has built Excel with some robust methods for importing external data into your workbooks, making Excel a terrific front-end application that analyzes data it does not need to store.

INTRODUCING ADO

ADO is an acronym for ActiveX Data Objects, which is the technology Microsoft recommends for accessing data in external databases. Excel's spreadsheets, being tabular row and column objects, share common features with database tables, providing a natural environment for data to be transferred between Excel and relational databases.

From Excel, using ADO you can do the following:

- Connect to most any external database in the Windows operating system, as long as that database has, as many do, an ODBC (Open Database Connectivity) or OLE DB (Object Linking and Embedding Database) driver.
- Add, delete, and edit records from a database to your workbook, or from your workbook to a database.
- Query data to return a recordset, enabling you to import some or all records from a database table directly to your worksheet, for whatever analysis you want to perform, just as if the data was already in Excel.

DEFINITIONS OF DATABASE TERMS

Because this lesson introduces concepts for external data access, it contains more descriptive theory about databases than actual code examples. In Lesson 32, you see several working examples of how Excel utilizes ADO and SQL in conjunction with Access databases. If you are unfamiliar with database terminology, the following definitions for common database terms might help you throughout this lesson.

A *database* is an organized collection of related information.

DAO (Data Access Objects) is a library of objects and their associated methods and properties that can be used to represent objects in databases, enabling Excel to interact directly with databases through VBA.

DBMS is an abbreviation for *database management system*. Popular examples of database management systems include dBASE, Paradox, and Microsoft Access.

A *field* is a column in a list such as in an Excel worksheet or Access database that describes a characteristic about records, such as first name or city.

ODBC (Open Database Connectivity) is a database standard that allows a program to connect to and manipulate a data source, enabling a single user to access many different databases.

A *primary key* is one or more fields that determine the uniqueness of each record in a database.

A *query* is a series of statements written in Structured Query Language to specify the tables and fields you want to work with that add, modify, remove, or return data from a database.

A *record* is a row of data in a table.

A *recordset* is one or more records (rows) of data derived from a table.

A *relational database* is a collection of data items organized as a set of formally described tables from which data can be accessed or reassembled in many ways.

NOTE Prior to ADO, Microsoft's primary recommended tool for accessing external data was an interface called DAO, or Data Access Objects. The DAO interface has become all but obsolete due to its limitations as compared to ADO, though DAO is still supported by ADO. The two technologies share many of the same code syntaxes but they are not the same in terms of flexibility and performance. You still do have a choice between the two, but you'll be much better served by ADO, which is why it is covered in this book.

With entire books devoted to database integration with ADO, there is much more complexity to the topic than this lesson is meant to cover. The best way to start becoming familiar with ADO is to examine the three primary tools in its object model: the Connection object, the Recordset object, and the Command object.

The Connection Object

The Connection object establishes a path that connects Excel and the database. With ADO from Excel, you normally issue commands that pass information back and forth through the Connection object. Among the key methods belonging to the Connection object are `Open`, which establishes the database connection, and `Close`, which closes the connection. The Connection object's `ConnectionString` property defines how to connect to the database.

You connect to the database with the `Provider` keyword. The following line of code is a common syntax for Excel versions 2007 through 2013:

```
Provider = "Microsoft.ACE.OLEDB.12.0;Data Source=_
C:\YourFilePath\Database1.accdb";Persist Security Info=False;"
```

In versions of Excel prior to 2007, the `Provider` would have been specified as the Microsoft Jet database engine of Access:

```
Provider = "Microsoft.Jet.OLEDB.4.0;" & _
"Data Source=C:\YourFilePath\Database1.accdb; Extended Properties=Excel 8.0;"
```

Or, depending on the circumstance, more simply:

```
Provider = "Microsoft.Jet.OLEDB.4.0"
```

NOTE When working with databases, you almost always connect to them, meaning you do not open them in the way you'd open a Word document if you were working with Word from Excel. The Connection object is like a conduit between Excel and your database.

The Recordset Object

The Recordset object is probably the most commonly used object in ADO. When you instruct ADO to retrieve a single record or the entire count of records from a database table, you use the Recordset object to do that.

Among the key members of the Recordset object are the following:

- The `ActiveConnection` property, which is a connection string or a Connection object that identifies the connection being used to access the database. As with this property for the Command object, where `objRecordset` and `objConnection` are object variables, the `ActiveConnection` syntax is

```
Set objRecordset.ActiveConnection = objConnection
```

- The Open method opens the Recordset object so you can access the data. Its syntax is
`Recordset.Open Source, ActiveConnection, CursorType, LockType, Options`
Note that the `Source` argument is often a string that names the table from which the recordset should be retrieved.
- The Close method closes an open Recordset object. With the Recordset object declared as `dbRecordset`, the syntax for Close would be
`dbRecordset.Close`

The Command Object

The Command object holds information about the kind of task being run, which is usually related to action queries in Access, or procedures in SQL, which are described in the next section. A Command object can also return a list of data records, and is most often run with a combination of parameters, of which there are more than this lesson can possibly cover.

The Command object has three important properties:

- The ActiveConnection property, which, like the ActiveConnection property for the Recordset object, is a connection string or a Connection object that identifies the connection being used to access the database. For example, this syntax assigns a Connection object to the ActiveConnection property, where `objRecordset` and `objConnection` are object variables:
`Set objRecordset.ActiveConnection = objConnection`
- The CommandText property, which sets the command that will be executed by the database and will usually be an SQL string.
- The CommandType property, which tells the database how to interpret and execute the CommandText's instructions.

AN INTRODUCTION TO STRUCTURED QUERY LANGUAGE (SQL)

Structured Query Language (SQL) is a database language used for querying, updating, and managing relational databases. SQL is used to communicate with the vast majority of databases that are commonly in use today.

SQL is a complex language in response to the rigid nature of table design in relational database construction. This lesson covers SQL's four basic operations of `SELECT`, `INSERT`, `UPDATE`, and `DELETE`. As a reminder of what I mentioned at the beginning of this lesson, you'll find several examples of these operations in Lesson 32 that show how to work with Access from Excel.

NOTE Notice that SQL statements, such as SELECT and INSERT, are shown in uppercase. This is a standard SQL programming practice and a good habit to get into from the start. The SQL code examples in this book are relatively small, but SQL code can be very large and complex. SQL is easier to read when its statements are shown in uppercase, which distinguishes them from the clauses of code with which they are associated.

The SELECT Statement

The SELECT statement retrieves data in the form of one or more rows (records) from one or more tables. The SELECT statement is probably SQL's most commonly used operation, because it tells the data source what field(s) you want to return from what table(s).

If you want to retrieve all columns and all rows from the Vendors table, the expression in SQL is as follows:

```
SELECT *
FROM Vendors
```

Sometimes you might not want to retrieve all columns. The following example retrieves the State column from the Vendors table, if you want to know the count of your vendors per state:

```
SELECT State
FROM Vendors
```

If you want to see a list of vendors and the names of their contact people, but only for vendors in California, the following example accomplishes that. Note that the literal string criterion California is in single quotes, which is SQL's required syntax:

```
SELECT VendorName, ContactName
FROM Vendors
WHERE State 'California'
```

If you want to retrieve the previous recordset by having it already sorted by the VendorName field, you could add the ORDER BY statement and specify the field name as follows:

```
SELECT VendorName, ContactName
FROM Vendors
WHERE State 'California'
ORDER BY VendorName
```

The INSERT Statement

The INSERT statement adds a new row (record) to a table. You need to specify the name of the table where the row will be added. You can optionally omit the field names from the INSERT statement, but it is advisable that you name them anyway because it helps you to see that the values you are entering are in the same order as the field names.

An example of using `INSERT` is this fictional pair of statements that respectively place the values 5432, Doe, John, Male into a table named `Employees`, for fields named `EmployeeID`, `LastName`, `FirstName`, and `Gender`:

```
INSERT INTO Employees (EmployeeID, LastName, FirstName, Gender)
VALUES ('5432', 'Doe', 'John', 'Male')
```

NOTE *It's standard SQL programming practice to enter the statements in uppercase. It is mandatory SQL programming practice to place the string literal VALUES within single quotes, just as you see it here.*

If you had opted to enter the preceding SQL code without naming each field, the syntax example for that same procedure would have been as follows:

```
INSERT INTO Employees
VALUES ('5432', 'Doe', 'John', 'Male')
```

The UPDATE Statement

The `UPDATE` statement enables you to change the values in one or more columns (fields) in a table. `UPDATE` is most commonly used to modify the value of a specific record that you identify with the `WHERE` clause. You also need to specify each column you want to change, and what each column's new value should be.

The following example shows how you could update the contact name of one of your company's vendors in the `ContactName` column of the `Vendors` table. You need to be careful to specify the `WHERE` clause so that only one record is changed, and that it is the correct record.

In the `Vendors` table, you have a field named `VendorID` that lists unique vendor identification numbers. The vendor name itself is Widgets, Inc., but that is not as important as its vendor identification number. Suppose that the vendor identification number for Widgets, Inc. is 1234. The new contact name is John Doe, executed with these three statements in SQL:

```
UPDATE Vendors
SET ContactName = 'John Doe'
WHERE VendorID = '1234'
```

If the `ContactName` field had many empty (referred to as Null) values, and you wanted to fill those empty spaces with the word Unknown, the following example would accomplish that:

```
UPDATE Vendors
SET ContactName = 'Unknown'
WHERE ContactName IS NULL
```

The DELETE Statement

The `DELETE` statement deletes one or more rows from a table. If you want to delete the vendor named Widgets, Inc., you would use the `WHERE` statement to specify which value in which column should identify the record for Widgets, Inc. The `VendorID` column is the perfect column for this task because a large company might have two vendors with the same name.

The following SQL statements would delete the record from the Vendors table that has the value 1234 in the VendorID column:

```
DELETE FROM Vendors
WHERE VendorID = '1234'
```

NOTE Make absolutely certain you specify the WHERE clause, because if you do not, every row from the Vendors table would be deleted. If an empty table is what you want, this fictional sequence would accomplish that:

```
DELETE FROM Vendors
```

Odds are, you don't want an empty table with all rows deleted from it. The kicker is that after the rows are deleted, you cannot undo that action as you can in Excel. Unless you are good friends with an experienced database programmer who might (or might not) be able to recover your unintentionally deleted rows, take heed and always specify the WHERE clause in your SQL DELETE actions.

TRY IT

This lesson introduced the fundamentals of ADO and SQL. You see several examples in Lesson 32 of VBA macros that show how to program ADO with SQL to interact with Access databases from Excel.

Here is a way to get a head start on the instruction in Lesson 32 to become familiar with database tables. Open Access and create a new database. Create a new table and enter some fictional data such as a mailing list with fields for FirstName, LastName, StreetAddress, City, State, Country, and Postal Code. Make a dozen or so entries and get a feel for navigating and editing a database table. For example, Figure 27-1 shows a table in Access being populated with hypothetical employee information, such as you might see in a company's personnel database.

ID	Field1	Field2	Field3	Field4	Click to Add
5 1234	EmployeeID	Last Name	First Name	Title	
6 6548	Doe	Jane	CFO		
7 5421	Smith	John	Sales Manager		
8 8447	Lee	Mary	Bookkeeper		
9 2133	Jones	Joe	Analyst		
*	(New)	Jackson	Lisa	Receptionist	

FIGURE 27-1

You'll notice an important distinction between an Access table and an Excel worksheet. Database tables do not have row headers as numbers, or columns designated by letters. Columns (called *fields* in a database environment) rely on being identified by their field headers such as FirstName, LastName, and so on. Rows (called *records*) rely on being identified by one or more key fields, or certain properties of other fields such as being empty (*Null*) or having date entries between a start date and an end date.

You might also want to surf the Web for sites that list SQL objects and their associated properties and methods. Keep in mind that SQL's capacity for database interaction goes far beyond what you'll need it to do for your Excel projects, so stick with the basics for now when perusing SQL instructional material.

REFERENCE *There is no video or code download to accompany this lesson.*

28

Impressing Your Boss (or at Least Your Friends)

Microsoft estimates that Excel is loaded onto some 600 million computers worldwide. One trait all Excel users have in common is that no one knows all there is to know about Excel. The power and diversity of Excel's native capabilities alone are more than enough to master. With VBA for Excel—each new version having more features than the one before—the capabilities for performance, object programming, and data management are virtually limitless.

The theme of this lesson is to show a variety of examples of what Excel can achieve with VBA. I encourage you to continue advancing your VBA skills after reading this book, and hopefully, being inspired by the more advanced examples in this lesson.

NOTE *In general, the examples in this lesson are a bit more advanced than what you've seen in the book so far. Be sure to watch the 15 videos of advanced VBA examples that accompany this book!*

SELECTING CELLS AND RANGES

A common request I have received from Excel users is how to show the current location on a worksheet by highlighting the active cell, row, or column. It is easier to maintain your bearings in worksheets such as budgets and financial statements when a color stands out to show where you are.

Coloring the Active Cell, Row, or Column

In Figure 28-1, three examples are shown that format either the active cell only, the active cell's entire row and column, or the row and column within the active cell's current region.

Active cell

	A	B	C	D	E
1	Widgets, Inc.				
2	2015 Budget				
3					
4	Q1	Q2	Q3	Q4	
5	Income	893	651	757	462
6	Expenses	849	466	556	444
7	Net Gain	44	185	201	18
8	Capital	24	100	188	16
9	Net Profit	20	85	13	2
10					
11					
12	Month	Name	Sales		
13	January	Bill	903		
14	February	Bob	959		
15	March	Tom	707		
16	April	Mike	618		
17	May	Jim	796		
18	June	Sophia	550		
19	July	Angie	49		
20	August	Bella	523		
21	September	Frank	654		
22	October	Joe	919		
23	November	Pete	532		
24	December	Alex	727		
25					

Active cell's row and column

	A	B	C	D	E
1	Widgets, Inc.				
2	2015 Budget				
3					
4	Q1	Q2	Q3	Q4	
5	Income	893	651	757	462
6	Expenses	849	466	556	444
7	Net Gain	44	185	201	18
8	Capital	24	100	188	16
9	Net Profit	20	85	13	2
10					
11					
12	Month	Name	Sales		
13	January	Bill	903		
14	February	Bob	959		
15	March	Tom	707		
16	April	Mike	618		
17	May	Jim	796		
18	June	Sophia	550		
19	July	Angie	49		
20	August	Bella	523		
21	September	Frank	654		
22	October	Joe	919		
23	November	Pete	532		
24	December	Alex	727		
25					

Active cell's row and col current region

	A	B	C	D	E	F
1	Widgets, Inc.					
2	2015 Budget					
3						
4	Q1	Q2	Q3	Q4		
5	Income	893	651	757	462	
6	Expenses	849	466	556	444	
7	Net Gain	44	185	201	18	
8	Capital	24	100	188	16	
9	Net Profit	20	85	13	2	
10						
11						
12	Month	Name	Sales			
13	January	Bill	903			
14	February	Bob	959			
15	March	Tom	707			
16	April	Mike	618			
17	May	Jim	796			
18	June	Sophia	550			
19	July	Angie	49			
20	August	Bella	523			
21	September	Frank	654			
22	October	Joe	919			
23	November	Pete	532			
24	December	Alex	727			
25						

FIGURE 28-1

These are `Worksheet_SelectionChange` events. To install this behavior for a worksheet, right-click that worksheet tab, select View Code, and paste either of the following procedures (but not more than one at a time per worksheet) into the large white area that is the worksheet module. Press Alt+Q to return to the worksheet. Then, select a few cells to see the effects of the code.

To format the active cell only:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
Application.ScreenUpdating = False
Cells.Interior.ColorIndex = 0
Target.Interior.Color = vbCyan
Application.ScreenUpdating = True
End Sub
```

To format the entire row and column of the active cell:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
If Target.Cells.Count > 1 Then Exit Sub
Application.ScreenUpdating = False
Cells.Interior.ColorIndex = 0
With Target
.EntireColumn.Interior.Color = vbCyan
.EntireRow.Interior.Color = vbCyan
End With
Application.ScreenUpdating = True
End Sub
```

To format the row and column within the current region of the active cell:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
Cells.Interior.ColorIndex = 0
If IsEmpty(Target) Or Target.Cells.Count > 1 Then Exit Sub
Application.ScreenUpdating = False
With ActiveCell
Range(Cells(.Row, .CurrentRegion.Column), _
Cells(.Row, .CurrentRegion.Columns.Count + .CurrentRegion.Column - 1)) _
.Interior.Color = vbCyan
Range(Cells(.CurrentRegion.Row, .Column), _
Cells(.CurrentRegion.Rows.Count + .CurrentRegion.Row - 1, .Column)) _
.Interior.Color = vbCyan
End With
Application.ScreenUpdating = True
End Sub
```

Coloring the Current and Prior Selected Cells

This section explains how you can highlight not only the current cell but also the cell you selected before you selected your current cell. To make it easy to distinguish between the two cells, the currently selected cell is colored cyan, and the prior selected cell is colored magenta.

In Figure 28-2, cell C5 is the active (currently selected) cell, indicated by its cyan color when you install the following code into your workbook. You can also see its address in the address bar. Before the image of Figure 28-2 was created, cell H12 had been selected, evidenced by its magenta color.

Atlas Widgets Company Budget													
	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
3	Income	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
4	Sales	3,495	8,044	2,906	7,409	2,772	3,874	5,081	5,852	3,065	9,648	8,214	8,938
5	Consulting	3,109	8,313	9,111	8,961	7,999	5,248	8,065	3,478	9,456	6,584	6,847	1,935
6	Training	4,665	3,675	6,578	5,599	5,458	8,334	377	6,124	7,251	6,245	7,154	7,124
7	Total	11,269	20,032	18,595	21,969	16,229	17,456	13,523	15,454	19,772	22,477	22,215	17,997
8													
9	Expenses	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
10	Office rent	2,002	791	381	3,667	3,124	696	501	3,089	2,855	930	3,404	1,080
11	Payroll	689	2,007	2,646	1,012	1,541	1,253	1,399	1,212	1,282	2,158	3,717	2,487
12	Utilities	21	1,681	2,988	3,653	1,254	3,411	3,748	812	1,357	3,618	1,234	1,410
13	Supplies	707	2,898	2,480	959	814	2,773	1,841	130	343	3,139	2,732	41
14	Shipping	1,035	1,843	2,085	155	2,108	1,850	1,688	2,126	2,182	3,652	2,283	191
15	Telephone	2,127	2,433	2,348	3,148	621	1,725	688	2,667	2,176	3,117	2,497	2,280
16	Marketing	3,029	108	2,283	1,569	2,714	1,374	954	1,523	3,633	3,436	337	1,546
17	Taxes	1,211	1,095	2,288	2,066	3,534	1,750	1,079	1,159	2,237	159	3,388	3,065
18	Total	10,821	12,857	17,499	16,228	15,709	14,833	11,899	12,717	16,065	20,209	19,593	12,100
19													
20	Net Gain	448	7,175	1,096	5,741	520	2,623	1,624	2,737	3,707	2,268	2,622	5,897

FIGURE 28-2

In Figure 28-3, the currently selected cell is L18, colored cyan. Now cell C5, which was selected before as seen in Figure 28-2, is colored magenta.

L18	Atlas Widgets Company Budget												
	A	B	C	D	E	F	G	H	I	J	K	L	M
1													
2													
3	Income	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
4	Sales	3,495	8,044	2,906	7,409	2,772	3,874	5,081	5,852	3,065	9,648	8,214	8,938
5	Consulting	3,109	8,313	9,111	8,961	7,999	5,248	8,065	3,478	9,456	6,584	6,847	1,935
6	Training	4,665	3,675	6,578	5,599	5,458	8,334	377	6,124	7,251	6,245	7,154	7,124
7	Total	11,269	20,032	18,595	21,969	16,229	17,456	13,523	15,454	19,772	22,477	22,215	17,997
8	Expenses	Jan	Feb	Mar	Apr	May	Jun	Jul	Aug	Sep	Oct	Nov	Dec
10	Office rent	2,002	791	381	3,667	3,124	696	501	3,089	2,855	930	3,404	1,080
11	Payroll	689	2,007	2,646	1,012	1,541	1,253	1,399	1,212	1,282	2,158	3,717	2,487
12	Utilities	21	1,681	2,988	3,653	1,254	3,411	3,748	812	1,357	3,618	1,234	1,410
13	Supplies	707	2,898	2,480	959	814	2,773	1,841	130	343	3,139	2,732	41
14	Shipping	1,035	1,843	2,085	155	2,108	1,850	1,688	2,126	2,182	3,652	2,283	191
15	Telephone	2,127	2,433	2,348	3,148	621	1,725	688	2,667	2,176	3,117	2,497	2,280
16	Marketing	3,029	108	2,283	1,569	2,714	1,374	954	1,523	3,633	3,436	337	1,546
17	Taxes	1,211	1,095	2,288	2,066	3,534	1,750	1,079	1,159	2,237	159	3,388	3,065
18	Total	10,821	12,857	17,499	16,228	15,709	14,833	11,899	12,717	16,065	20,209	19,533	12,100
19	Net Gain	448	7,175	1,096	5,741	520	2,623	1,624	2,737	3,707	2,268	2,622	5,897

FIGURE 28-3

The following procedure that produces this functionality is a Selection_Change event. Place it into your worksheet module and test the code by selecting a few cells:

```
Private Sub Worksheet_SelectionChange(ByVal Target As Range)
    Cells.Interior.ColorIndex = 0
    Static PriorCell As Range
    If Not PriorCell Is Nothing Then _
        PriorCell.Interior.Color = vbMagenta
    Target.Interior.Color = vbCyan
    Set PriorCell = Target
End Sub
```

FILTERING DATES

When it comes to filtering dates, a little VBA goes a long way in dealing with the nemesis of seemingly countless different formats in which a date can be represented in Excel. The key to filtering dates is to treat them as the numeric value they are, and to use the `DateSerial` function for an unambiguous date reference. No matter what the date formatting gods throw at you, the following macros filter your dates.

Filtering between Dates

On the left in Figure 28-4, dates are shown in many formats in column A. To make it more challenging, cells B2 and B3 contain the start and end date criteria that are formatted the same as only one cell in the list being filtered. The macro named `FilterBetweenDates` filters the dates as shown on the right in Figure 28-4.

Before filter, with dates in various formats.		After filter.	
A	B	A	B
1	Filter Criteria For Dates	1	Filter Criteria For Dates
2	Start Date	2	Start Date
3	End Date	3	End Date
4		4	
5	Date	5	Date
6	10/26/2015	12	Nov 13, 2015
7	October 29, 2015	13	11/16/2015
8	1-Nov-2015	14	Nov 19 15
9	Nov 4, 2015	15	221115
10	November 7, 2015	16	November 25, 2015
11	101115	17	28-Nov-2015
12	Nov 13, 2015	18	Dec 1, 2015
13	11/16/2015	19	12/4/2015
14	Nov 19 15	20	071215
15	221115	21	12/10/2015
16	November 25, 2015	22	
17	28-Nov-2015	23	
18	Dec 1, 2015	24	
19	12/4/2015	25	
20	071215	26	
21	12/10/2015		
22	December 13, 2015		
23	16-Dec-2015		
24	Dec 19, 2015		
25	1/1/2016		
26			

FIGURE 28-4

```

Sub FilterBetweenDates()
    Application.ScreenUpdating = False
    ActiveSheet.AutoFilterMode = False

    Dim StartDate As Date, EndDate As Date
    Dim FilterStartDate As Date, FilterEndDate As Date
    Dim LastRow As Long
    Dim FilterRange As Range

    StartDate = Range("B2").Value
    EndDate = Range("B3").Value
    LastRow =
        Cells.Find(What:="*", After:=Range("A1"),
        SearchOrder:=xlByRows, SearchDirection:=xlPrevious).Row
    Set FilterRange = Range("A5:A" & LastRow)
    FilterStartDate =
        DateSerial(Year(StartDate), Month(StartDate), Day(StartDate) - 1)
    FilterEndDate =
        DateSerial(Year(EndDate), Month(EndDate), Day(EndDate) + 1)

    FilterRange.AutoFilter _
        Field:=1, Criteria1:=">" & CDbL(FilterStartDate), _
        Operator:=xlAnd, _

```

```

Criteria2:="<" & CDbl(FilterEndDate)

Set FilterRange = Nothing
Application.ScreenUpdating = True
End Sub

```

Filtering for Dates before Today's Date

The macro named `FilterDateBeforeToday` filters for dates before today's date. The reference to where the data table begins is the same as what is shown in Figure 28-4.

```

Sub FilterDateBeforeToday()
Application.ScreenUpdating = False
ActiveSheet.AutoFilterMode = False
Dim LastRow As Long, FilterRange As Range
LastRow =
Cells.Find(What:="*", After:=Range("A1"), _
SearchOrder:=xlByRows, SearchDirection:=xlPrevious).Row
Set FilterRange = Range("A5:A" & LastRow)
FilterRange.AutoFilter Field:=1, Criteria1:="<" & CDbl(Date)
Set FilterRange = Nothing
Application.ScreenUpdating = True
End Sub

```

Filtering for Dates after Today's Date

The macro named `FilterDateAfterToday` filters for dates after today's date. The reference to where the data table begins is the same as what is shown in Figure 28-4.

```

Sub FilterDateAfterToday()
Application.ScreenUpdating = False
ActiveSheet.AutoFilterMode = False
Dim LastRow As Long, FilterRange As Range
LastRow = Cells.Find(What:="*", After:=Range("A1"), _
SearchOrder:=xlByRows, SearchDirection:=xlPrevious).Row
Set FilterRange = Range("A5:A" & LastRow)
FilterRange.AutoFilter Field:=1, Criteria1:=">" & CDbl(Date)
Set FilterRange = Nothing
Application.ScreenUpdating = True
End Sub

```

Deleting Rows for Filtered Dates More Than Three Years Ago

The macro named `DeleteRows3YearsOld` filters for dates that are three years ago from today's date:

```

Sub DeleteRows3YearsOld()
Application.ScreenUpdating = False
ActiveSheet.AutoFilterMode = False
Dim FilterRange As Range, myDate As Date
myDate = DateSerial(Year(Date) - 3, Month(Date), Day(Date))
Set FilterRange =
Range("A5:A" & Cells(Rows.Count, 1).End(xlUp).Row)
FilterRange.AutoFilter Field:=1, Criteria1:="<" & CDbl(myDate)
On Error Resume Next
With FilterRange
.Offset(1).Resize(.Rows.Count - 1).SpecialCells(xlCellTypeVisible).EntireRow.Delete
End With

```

```

Err.Clear
Set FilterRange = Nothing
ActiveSheet.AutoFilterMode = False
Application.ScreenUpdating = True
End Sub

```

SETTING PAGE BREAKS FOR SPECIFIED AREAS

If your worksheet has areas of data that you want to print on separate pages, you can establish page breaks based on a wide choice of cell properties or text values. With the following macro named PageBreakInsert, page breaks are set below each cell in column A that starts with Total, as shown in Figure 28-5.

```

Sub PageBreakInsert()
Cells.PageBreak = xlPageBreakNone
Dim cell As Range
For Each cell In Columns(1).SpecialCells(xlCellTypeConstants)
If Left(cell.Value, 5) = "Total" Then
With ActiveSheet
.HPageBreaks.Add Cells(cell.Row + 1, 1)
.DisplayAutomaticPageBreaks = True
End With
End If
Next cell
End Sub

```

Before, without page breaks					After, with page breaks below "Total..."						
A	B	C	D	E	A	B	C	D	E		
1	Atlas Widgets, Inc., Budget Summary (in thousands)				1	Atlas Widgets, Inc., Budget Summary (in thousands)					
2					2						
3	Income	Q1	Q2	Q3	Q4	3	Income	Q1	Q2	Q3	Q4
4	Widget sales	\$ 661	\$ 534	\$ 854	\$ 498	4	Widget sales	\$ 661	\$ 534	\$ 854	\$ 498
5	Widget rentals	\$ 334	\$ 153	\$ 118	\$ 205	5	Widget rentals	\$ 334	\$ 153	\$ 118	\$ 205
6	Total Income					6	Total Income				
7					7						
8	Payroll				8	Payroll					
9	Manager	\$ 10,000	\$ 10,000	\$ 10,000	\$ 11,500	9	Manager	\$ 10,000	\$ 10,000	\$ 10,000	\$ 11,500
10	Sales	\$ 9,300	\$ 9,300	\$ 9,300	\$ 97,400	10	Sales	\$ 9,300	\$ 9,300	\$ 9,300	\$ 97,400
11	Administrative					11	Administrative				
12	Total Payroll					12	Total Payroll				
13					13						
14	Maintenance					14	Maintenance				
15	Repairs	\$ 693	\$ 315	\$ 131	\$ 199	15	Repairs	\$ 693	\$ 315	\$ 131	\$ 199
16	Roofing	\$ 199	\$ 139	\$ 187	\$ 104	16	Roofing	\$ 199	\$ 139	\$ 187	\$ 104
17	Gutters	\$ 70	\$ 971	\$ 716	\$ 836	17	Gutters	\$ 70	\$ 971	\$ 716	\$ 836
18	HVAC	\$ 123	\$ 252	\$ 308	\$ 896	18	HVAC	\$ 123	\$ 252	\$ 308	\$ 896
19	Total Maint					19	Total Maint				
20					20						
21	Supplies					21	Supplies				
22	Notebooks	\$ 389	\$ 359	\$ 752	\$ 332	22	Notebooks	\$ 389	\$ 359	\$ 752	\$ 332
23	Pens	\$ 14	\$ 18	\$ 22	\$ 31	23	Pens	\$ 14	\$ 18	\$ 22	\$ 31
24	Coffee	\$ 104	\$ 247	\$ 167	\$ 233	24	Coffee	\$ 104	\$ 247	\$ 167	\$ 233
25	Printer toner	\$ 125	\$ 248	\$ 137	\$ 122	25	Printer toner	\$ 125	\$ 248	\$ 137	\$ 122
26	Total Supplies					26	Total Supplies				
27					27						
28	Taxes					28	Taxes				
29	Property	\$ 225	\$ 225	\$ 225	\$ 225	29	Property	\$ 225	\$ 225	\$ 225	\$ 225
30	Income	\$ 250	\$ 250	\$ 250	\$ 250	30	Income	\$ 250	\$ 250	\$ 250	\$ 250
31	State	\$ 100	\$ 100	\$ 100	\$ 100	31	State	\$ 100	\$ 100	\$ 100	\$ 100

FIGURE 28-5

USING A COMMENT TO LOG CHANGES IN A CELL

This section shows how you can keep a running log of changes to a cell's text. Suppose you want your employees to enter an explanation or description into a cell regarding a topic on your spreadsheet. Maybe there's a new product being developed and you'll utilize cell A1 for team members to enter their ideas during production. You want to keep a record of everything entered, without burdening anyone with how to edit existing text or how to add a new comment to a cell.

In Figure 28-6, new entries are made into cell A1 on an ongoing basis. Although each new entry overrides preexisting text, the following procedure captures all the text that has been previously entered. There's also a date and time stamp for each new entry, and an empty line between entries in the comment for readability. This is a `Worksheet_Change` procedure, which goes into your worksheet module:

```
Private Sub Worksheet_Change(ByVal Target As Range)
With Target
If .Address <> "$A$1" Then Exit Sub
If IsEmpty(Target) Then Exit Sub
Dim strNewText$, strCommentOld$, strCommentNew$
strNewText = .Text
If Not .Comment Is Nothing Then
    strCommentOld = .Comment.Text & Chr(10) & Chr(10)
Else
    strCommentOld = ""
End If
On Error Resume Next
.Comment.Delete
Err.Clear
.AddComment
.Comment.Visible = False
.Comment.Text Text:=strCommentOld &
Format(VBA.Now, "MM/DD/YYYY at h:MM AM/PM") & Chr(10) & strNewText
.Comment.Shape.TextFrame.AutoSize = True
End With
End Sub
```

The figure consists of three vertically stacked screenshots of an Excel spreadsheet. Each screenshot shows a table with columns A, B, C, and D. Row 1 contains the text 'This is the first entry.' in cell A1. Row 2 is empty. Row 3 is empty. In the second screenshot, row 1 still contains 'This is the first entry.', but cell B1 now contains the timestamp '12/12/2014 at 8:39 AM' and the text 'This is the first entry.' in a comment box. Row 4 is empty. Row 5 is empty. In the third screenshot, row 1 still contains 'This is the first entry.', but cell B1 now contains the timestamp '12/12/2014 at 8:39 AM'. Row 2 contains the timestamp '12/13/2014 at 10:26 PM' and the text 'This is the second entry.' in a comment box. Row 3 is empty. Row 4 is empty. Row 5 is empty. Row 6 contains the timestamp '12/13/2014 at 10:26 PM' and the text 'This is the second entry.' in a comment box. Row 7 is empty. Row 8 contains the timestamp '12/14/2014 at 2:47 PM' and the text 'Third entry with more text to show the comment's Autosize property.' in a comment box. Row 9 is empty.

	A	B	C	D
1	This is the first entry.	12/12/2014 at 8:39 AM This is the first entry.		
2				
3				
	A	B	C	D
1	This is the second entry.	12/12/2014 at 8:39 AM This is the first entry.		
2				
3				
4				
5		12/13/2014 at 10:26 PM This is the second entry.		
	A	B	C	D
1	Third entry with more text to show the comment's Autosize property.	12/12/2014 at 8:39 AM This is the first entry.		
2				
3				
4				
5		12/13/2014 at 10:26 PM This is the second entry.		
6				
7		12/14/2014 at 2:47 PM Third entry with more text to show the comment's Autosize property.		
8				
9				

FIGURE 28-6

USING THE WINDOWS API WITH VBA

With the Windows API (application programming interface), you can program Windows objects that are not specific to Excel. Examples of Windows objects are the browser window, the status bar, and, as the following two macros demonstrate, the clipboard and the recycle bin.

NOTE Starting in version 2010 and continuing with version 2013, you can install Excel as a 64-bit application if you are running a 64-bit version of Windows. Many Excel users, including myself, prefer the 32-bit version because it provides all the power needed while supporting ActiveX controls. Other Excel users prefer the 64-bit version if they work with enormous amounts of data.

The examples in this section are 32-bit API declarations and might not work in 64-bit versions. This raises the larger point that if your workbooks will be shared among both versions, your code must be compatible for either version to run it.

In most cases, your 32-bit API declarations will be compatible with 64-bit versions by inserting `PtrSafe` after the `Declare` key word. Fortunately, you don't need to create two workbooks, but you do need to declare your API functions twice, using an `If...Then...Else` statement to establish the API calls for both versions. Lesson 32 shows this construction for an example that opens an Access database file.

The introduction of 64-bit Excel is relatively new, and it can be difficult to remember the nuances, as well as the syntaxes. For example, versions of Excel before 2010, including version 2007, do not recognize the `PtrSafe` keyword.

For an excellent resource about this topic, Jan Karel Pieterse of JKP Application Development Services (<http://www.jkp-ads.com>) maintains an ongoing list of proper syntax for API declarations in 32-bit and 64-bit versions. You can visit Jan Karel's web page at <http://www.jkp-ads.com/articles/apideclarations.asp>.

Clearing the Clipboard

The Windows clipboard is a temporary storage area for information that you have copied or moved from one place and plan to use somewhere else. You cannot see or touch the clipboard but you can work with it to copy, cut, paste, and clear data.

You can copy some 30 types of data onto your clipboard beyond just text and formulas, such as graphics, charts, and hyperlinks. To truly empty the clipboard requires more than just pressing the Esc key or executing the VBA statement `Application.CutCopyMode = False`.

With the Windows API, the macro named `ClearClipboard` clears all data types on your clipboard. The API function calls that precede the macro go at the top of your module, above and outside of the macro itself:

```
Public Declare Function OpenClipboard Lib "user32" _
(ByVal hwnd As Long) As Long
Public Declare Function CloseClipboard Lib "user32" () As Long
Public Declare Function EmptyClipboard Lib "user32" () As Long
```

```
Sub ClearClipboard()
OpenClipboard (0&)
EmptyClipboard
CloseClipboard
End Sub
```

Emptying the Recycle Bin

This macro named `RecycleBinEmpty` empties the recycle bin. The API function call named `EmptyRecycleBin` goes at the top of your module, above and outside of the macro itself:

```
Declare Function EmptyRecycleBin _
Lib "shell32.dll" Alias "SHEmptyRecycleBinA" _
(ByVal hwnd As Long, _
ByVal pszRootPath As String, _
ByVal dwFlags As Long) As Long

Sub RecycleBinEmpty()
Dim rbEmpty As Long
rbEmpty = EmptyRecycleBin(0&, vbNullString, 1&)
End Sub
```

SCHEDULING YOUR WORKBOOK FOR SUICIDE

If you have developed a workbook that you want to self-expire by a certain date, such as a demonstration model or one that contains information or usefulness that will be outdated, you can program the workbook to delete itself. In this example, the workbook's suicide date is scheduled for December 31, 2015.

In actual practice, you might want to have a message box—say, seven days prior to the suicide date—to let the workbook's users know what to expect on the upcoming date of demise. You would also lock and password-protect the Visual Basic Editor to reduce the chance for the code to be altered or deleted.

Please be careful when employing this code. When it executes, the recycle bin is bypassed, so your workbook is gone forever. The code goes into the workbook module and is evaluated every time the workbook opens.

```
Sub Workbook_Open()
If Date <= #12/31/2015# Then Exit Sub
MsgBox "This workbook has expired.", vbExclamation, "Goodbye."
With ThisWorkbook
.Saved = True
.ChangeFileAccess xlReadOnly
Kill .FullName
.Close False
End With
End Sub
```

TRY IT

For this lesson, you establish data validation in a cell, for which the allowable entries are the items in a custom list. Data validation by itself cannot directly access custom lists, but with VBA you can establish data validation to access a custom list in your Excel application.

Custom lists are identified in VBA by their index number. In the collection of custom lists on my computer, a fifth one will be added and used for this example.

Lesson Requirements

If you have not already done so, please establish a fifth custom list in your Excel application. You probably already have four that came with your Excel version. Otherwise, you will need to edit the number 5 in Step 11 to a lower number representing an existing custom list that you prefer to use.

To get the sample workbook, you can download Lesson 28 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

The macro for this example uses the fifth custom list in an Excel application. If you are not familiar with custom lists, Steps 2 to 6 explain how to add a custom list.

You can add, delete, or edit the items in your custom list. When you run the macro again, those changes show in the data validation drop-down list.

Step-by-Step

1. Start by opening a new workbook.
2. If you are not familiar with adding a custom list, click the File tab and select the Options menu item as shown in Figure 28-7.

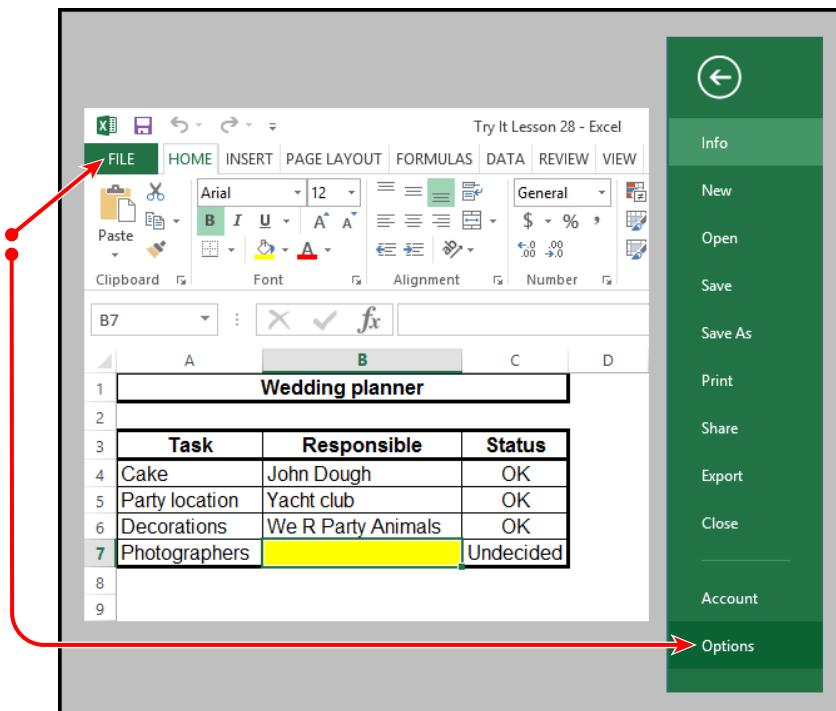


FIGURE 28-7

3. In the Excel Options dialog box, click the Advanced menu item. Scroll down to the General section, and click the Edit Custom Lists button as shown in Figure 28-8.

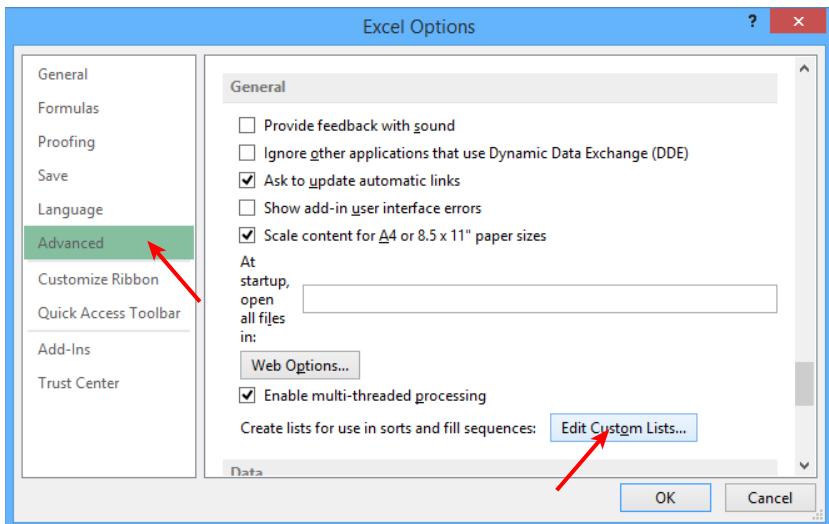


FIGURE 28-8

4. Click NEW LIST in the list box at the left, enter your list items in the list box at the right, and click Add as shown in Figure 28-9.

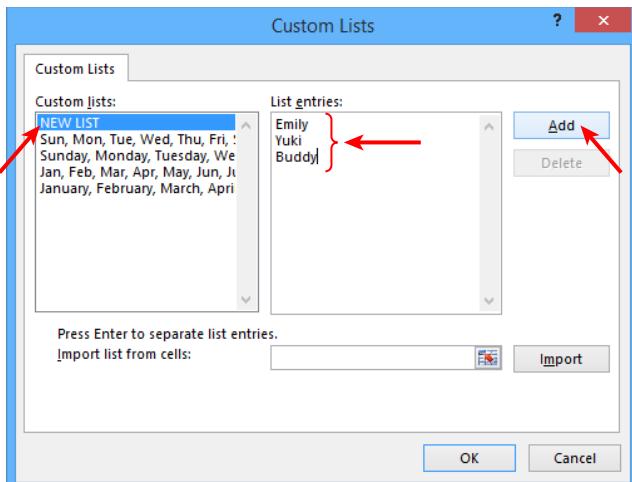


FIGURE 28-9

5. You see your new list of items in the list box at the left. Click OK as shown in Figure 28-10.

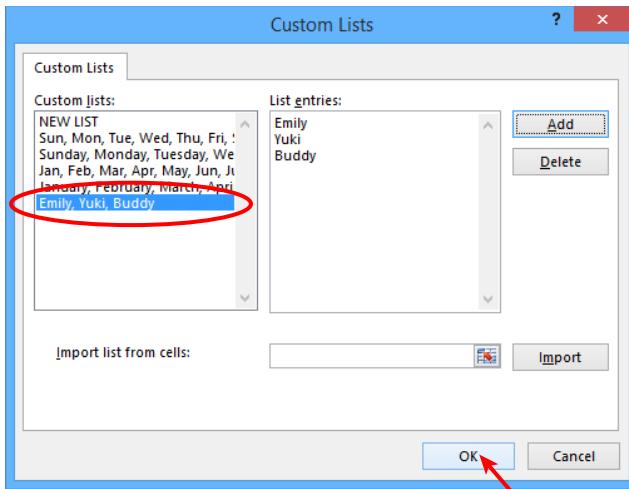


FIGURE 28-10

6. Click OK to exit the Excel Options dialog box as shown in Figure 28-11.

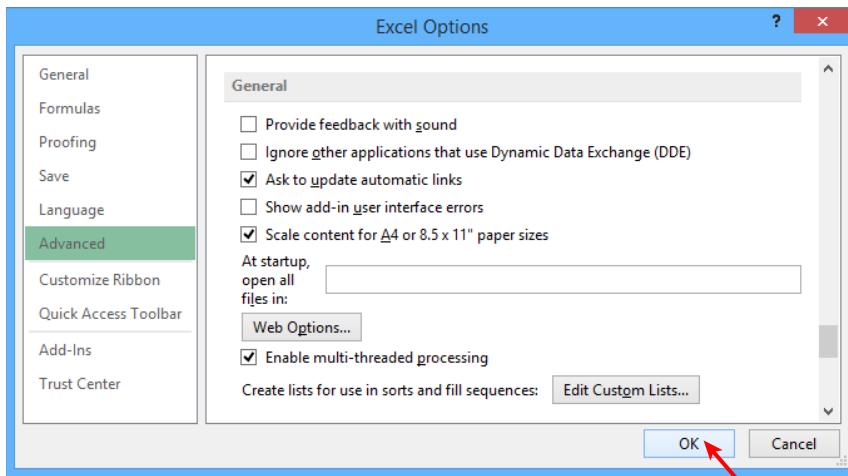


FIGURE 28-11

7. Press Alt+F11 to go to the Visual Basic Editor.
8. From the VBE menu bar, click Insert \Rightarrow Module.
9. In the module you just created, type `Sub CustomListDV` and press Enter. VBA automatically places a pair of empty parentheses at the end of the sub line, followed by an empty line, and the `End Sub` line below that. Your macro should look like this so far:

```
Sub CustomListDV()
```

```
End Sub
```

10. Declare a String type variable for custom items to be allowed by data validation, an Integer type variable to iterate through the array of items in your custom list, and a Variant type for the array itself:

```
Dim strCustomItems As String, intArray As Integer  
Dim myCustomList As Variant
```

11. Identify your custom list by its index number:

```
myCustomList = Application.GetCustomListContents(5)
```

12. Open a For...Next loop to iterate through each element in your custom list:

```
For intArray = LBound(myCustomList) To UBound(myCustomList)
```

13. Build the string for each custom item, separated by a comma:

```
strCustomItems = strCustomItems & myCustomList(intArray) & ","
```

14. Continue the loop until completion:

```
Next intArray
```

15. Delete the trailing comma after the last custom list item:

```
strCustomItems = Mid(strCustomItems, 1, Len(strCustomItems) - 1)
```

16. Establish data validation for the cell of interest:

```
With Range("B7").Validation  
'Delete the existing data validation.  
.Delete  
'Add the string of items from your custom list.  
.Add Type:=xlValidateList, _  
AlertStyle:=xlValidAlertStop, _  
Operator:=xlBetween, _  
Formula1:=strCustomItems  
'Error title if an invalid entry is attempted.  
.ErrorMessage = "Invalid entry!"  
'Error message if an invalid entry is attempted.  
'Note the ascii 10 character which is for a line break.  
.ErrorMessage = "Please enter an item" & Chr(10) & _  
"from the drop-down list."  
'Show the error icon in the message for invalid entries.  
.ShowError = True  
End With
```

```
End Sub
```

17. With your macro completed, press Alt+Q to return to the worksheet. To test the macro, press Alt+F8 to show the Macro dialog box. Select the macro named `CustomListDV` and click Run. Here is what the macro looks like in its entirety:

```

Sub CustomListDV()

'Declare variables:
'Custom items to be allowed by data validation,
'a counter for the array elements in the custom list,
'and your custom list.
Dim strCustomItems As String, intArray As Integer
Dim myCustomList As Variant

'Identify your custom list by its index number.
myCustomList = Application.GetCustomListContents(5)

'Loop through each element in your custom list.
For intArray = LBound(myCustomList) To UBound(myCustomList)
    'Build the string for each custom item, separated by a comma.
    strCustomItems = strCustomItems & myCustomList(intArray) & ","
    'Continue the loop until completion.
    Next intArray

'Delete the trailing comma after the last custom list item.
strCustomItems = Mid(strCustomItems, 1, Len(strCustomItems) - 1)

'Establish data validation for the cell(s) of interest.
With Range("B7").Validation
    'Delete the existing data validation.
    .Delete
    'Add the string of items from your custom list.
    .Add Type:=xlValidateList, _
        AlertStyle:=xlValidAlertStop, _
        Operator:=xlBetween, _
        Formula1:=strCustomItems
    'Error title if an invalid entry is attempted.
    .ErrorTitle = "Invalid entry !"
    'Error message if an invalid entry is attempted.
    'Note the ascii 10 character which is for a line break.
    .ErrorMessage = "Please enter an item" & Chr(10) & _
        "from the drop-down list."
    'Show the error icon in the message for invalid entries.
    .ShowError = True
End With

End Sub

```

REFERENCE Please select the video for Lesson 28 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

PART V

Interacting with Other Office Applications

- ▶ **LESSON 29:** Overview of Office Automation from Excel
- ▶ **LESSON 30:** Working with Word from Excel
- ▶ **LESSON 31:** Working with Outlook from Excel
- ▶ **LESSON 32:** Working with Access from Excel
- ▶ **LESSON 33:** Working with PowerPoint from Excel

29

Overview of Office Automation from Excel

As you may recall from Lesson 1, Visual Basic for Applications is a programming language created by Microsoft to automate operations in applications that support it, such as Excel. VBA is also the language that manipulates Microsoft Office applications in Access, Word, PowerPoint, and Outlook. So far, the focus of this book has been on running VBA from Excel, for the purpose of acting directly upon Excel in some way.

This section shows how to control other Office applications from Excel, using the same VBA programming language with which you are now familiar, but using a different set of methods and statements with which those other Office applications are familiar. The reasons for interacting with other Office applications might not be for the purpose of changing your Excel workbook application, but they will always be for the purpose of making your workbook projects more robust, versatile, and easier to use when the situation calls for it.

WHY AUTOMATE ANOTHER APPLICATION?

In the dawn of this modern era of personal computers, it was rare that two or more separate applications were able to communicate with each other. For two applications to share the same information, you usually had to retype the information manually into the other application that needed it. Today, thanks to the advances of drag and drop, and copy and paste, it has become a simple matter to share data across many applications.

The business of Excel is to perform calculations and analyze data. You can enter and edit text in Excel, but it is not a word processor. You can build data tables and compare their information, but Excel is not a relational database application. You can create charts and graphics in Excel but they cannot be presented in a sophisticated slideshow format. You can send a workbook through e-mail but Excel cannot manage your calendar or incoming e-mails the way an e-mail client can.

You get the idea—sooner or later you'll need to perform some kind of operation that another application was specially made to handle. This lesson lays the groundwork for you to understand Office automation from Excel, and the theory behind some best practices in doing so.

UNDERSTANDING OFFICE AUTOMATION

Where VBA is concerned, the only difference between Excel, Word, Access, PowerPoint, and Outlook lies in their object models. Each of these applications can access another's object model, so long as the target application has been properly installed on the host computer. Controlling one Office application from another becomes a simple matter of knowing how to link to the object model of the Office application you want to control.

The term “automation” is an Office programmer’s way of referring to the VBA technology that provides the ability to manipulate another application’s objects. Though VBA is the common language among Office applications, the respective object models differ in their objects’ names, methods, and properties. Both Excel and Word have a `Range` object but with different properties. Excel has a `Workbooks` object, which is the counterpart to PowerPoint’s `Presentations` object.

For Excel to access another Office application’s object model, a connection needs to be established to that target application. Two options for doing this exist: One option is called *early binding*, and the other option is called *late binding*. The term “binding” refers to the verification that an object exists, and that the command to manipulate that object’s methods and properties is valid.

Early Binding

With early binding, you establish a reference with the target application’s object library before you write your macro, so that the application’s objects, methods, and properties can be accessed in your code. For example, if you are using Office 2013 and you want to write a macro to open Word and edit a document, you would first need to establish a reference to the Microsoft Word 15.0 Object Library. To do that, you can go to the Visual Basic Editor, and from the menu bar click Tools \Rightarrow References. Scroll to select the reference and click OK, as shown in Figure 29-1.

NOTE VBA sees versions of Microsoft Office as numbers, not names. For example, VBA knows Office 2003 as version 11, Office 2007 as version 12, Office 2010 as version 14 (Microsoft knowingly skipped unlucky number 13), and Office 2013 as version 15. Therefore, if you are working with Office 2010 at home, you’d have Word 14 listed in your VBA References, but if you are using Office 2013 at work, you’d see Word 15 listed.

After you have established the proper reference, you can write a macro using early binding that will, for example, open a Word document in Office 2013. Suppose you already have a Word document named `myWordDoc.docx` that you keep in the path `C:\Your\File\Path\`. The following macro opens that document, using early binding:

```

Sub EarlyBindingTest()
    Dim wdapp As Word.Application, wddoc As Word.Document
    Set wdapp = New Word.Application
    wdapp.Visible = True
    Set wddoc = wdapp.Documents.Open(Filename:="C:\Your\File\Path\myWordDoc.docx")
End Sub

```

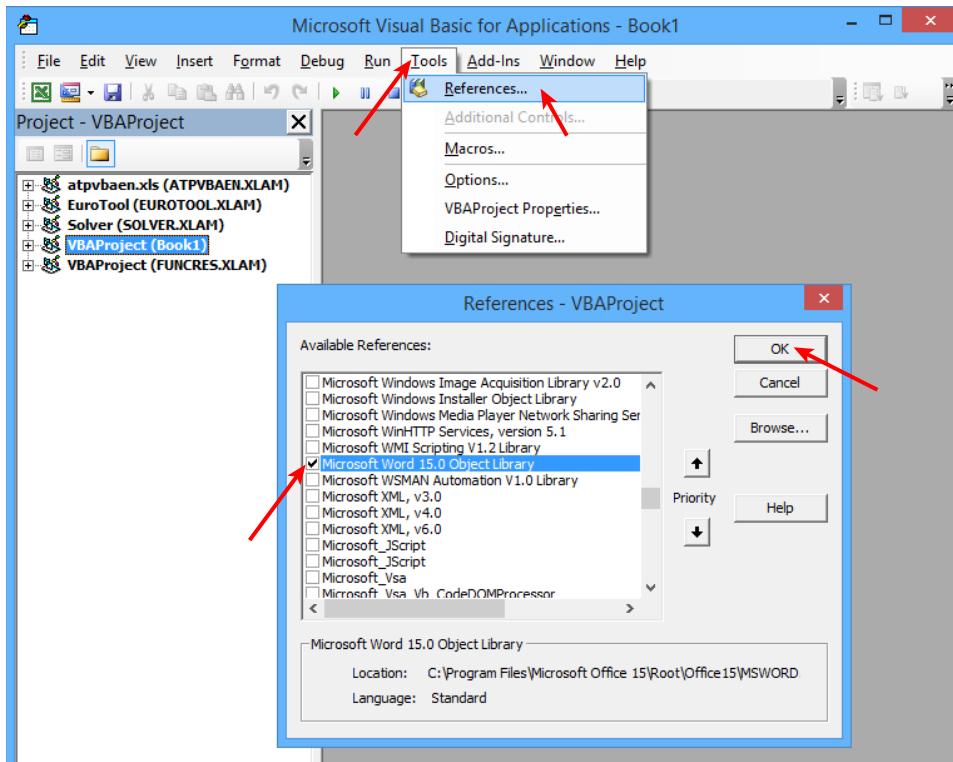


FIGURE 29-1

When you attempt to run this macro, you would immediately know if you did not properly establish the Word 15.0 library reference because you would be prompted by a compile error message, as shown in Figure 29-2.

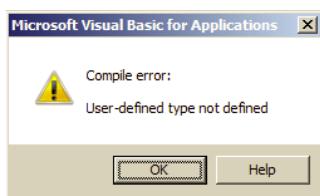


FIGURE 29-2

As you compose a macro using early binding, you will have the benefit of VBA's IntelliSense feature, where objects and properties pop up as you type your code's object references. And macros with

early binding run faster than macros performing the same task with late binding, because a reference has already been established to the target application's objects, methods, and properties.

NOTE *If your macro runs without errors but you don't see a Word document, or you don't even see Word on your taskbar, it could be that you really did create a new instance of Word, but it is not visible. In the Immediate window, type Word.Application.Visible = True and press Enter.*

So then, why would you ever not want to use early binding? Actually, there is a very good reason why not: The referenced object (Word 15.0 in this example) must exist on the computer. If it does not exist, an error occurs such as the one shown in Figure 29-2.

The concern is, unless you are composing your Office automation macros to be run on a system that you know for a fact will (a) be installed with the target application and (b) have the proper object library reference established in advance, chances are pretty good the macro will fail using early binding. And with new Office versions being released every few years, when you upgrade your Office version you need to edit all the macros in which you utilized early binding so that they don't refer to an outdated earlier version.

Late Binding

With late binding, you declare an object variable that refers to the target application, just as you would with early binding. However, instead of setting the variable to a specific (in this case) Word object, you create an object called a Word application.

If you use late binding, you do not use Tools \Rightarrow References to set a reference (as is required for early binding) because you do not know which Word object library version will be on a user's machine. Instead, you use code to create the object. The following macro named LateBindingTest accomplishes the same task as the EarlyBindingTest by opening a specific Word document:

```
Sub LateBindingTest()
    Dim WdApp As Object, wddoc As Object
    Set WdApp = CreateObject("Word.Application")
    WdApp.Visible = True
    Set wddoc = WdApp.Documents.Open(Filename:="C:\Your\File\Path\myWordDoc.docx")
End Sub
```

In a nutshell, when you declare a variable `As Object` and set it as `CreateObject`, VBA doesn't know whether the object is a cell, a worksheet, a Word application, or any other object. The code goes through a series of tests behind the scenes until it finds the correct application for the use intended by your code. That's the essential reason why late binding takes longer to execute.

Which One Is Better?

For my money, even with moderately sized macros, the extra seconds of run time due to late binding make up for the headaches of trying to accommodate every version of your target Office application, from 2000 through 2013. You will find that the VBA skills you are acquiring will lead to composing

macros that others will use, and you'll never know what Office versions are installed on users' systems. People have varying opinions on the merits of early versus late binding, so consider the pros and cons of both methods to decide which approach is best for you.

NOTE *I said that late binding code “takes longer” to execute than early binding. Depending on the task at hand, this should not dissuade you from using late binding. In fact, I use late binding exclusively in all my cross-application Office programming because of the benefits I mentioned.*

As you become more involved with programming, you'll find yourself identifying opportunities for efficiency in code execution. In the case of early versus late binding, or any set of programmable alternatives when the difference of execution is only a second or two, keep in mind that the project and its users are best served by code that gets the job done with minimal risk for error. No one notices an extra second or two of macro execution. Everyone notices runtime or performance errors.

TRY IT

In this lesson, you compose a macro using late binding that opens a presentation file in PowerPoint.

Lesson Requirements

For this lesson, you first create a PowerPoint presentation, name that file `PowerPointExample1`, and save it into the folder path `C:\Your\File\Path\`.

To get the sample Excel workbook and PowerPoint presentation files, you can download Lesson 29 from the book's website at www.wrox.com/go/excelvba24hour.

Hints

Late binding is a useful approach in cases like this, when the Office version is unknown.

If you want to refer to a sample PowerPoint presentation with a different name than `PowerPointExample1`, or a folder path other than `C:\Your\File\Path\`, be sure to modify those references in the following code.

Step-by-Step

1. Open a new workbook and press Alt+F11 to go to the Visual Basic Editor.
2. From the menu at the top of the VBE, click Insert ↴ Module.
3. In the module you just created, type `Sub openPowerPoint` and press Enter. VBA automatically places a pair of empty parentheses at the end of the `Sub` line, followed by an empty line, and the `End Sub` line below that. Your macro looks like this so far:

```
Sub OpenPowerPoint()
```

```
End Sub
```

4. Declare variables for the file path, the PowerPoint filename, and the file extension. The reason for the variable extension is that starting with Office version 2007, PowerPoint file extensions are commonly .pptx or .pptm. Prior to 2007, the extension for PowerPoint files was simply .ppt.

```
Dim myPath As String, myFileName As String, myExtension As String
```

5. Define the variables for myPath and myFileName:

```
myPath = "C:\Your\file\Path\"  
myFileName = "PowerPointExample1"
```

6. Use an If structure to define the extension string variable. Note the Val statement, which ensures the Office application version is regarded as a number for the logical evaluation of being less than or equal to version 11, which is Office 2003:

```
If Val(Application.Version) <= 11 Then  
    myExtension = ".ppt"  
Else  
    myExtension = ".pptx"  
End If
```

7. Declare the PowerPoint application object and set it using the CreateObject method for late binding:

```
Dim appPPT As Object  
Set appPPT = CreateObject("PowerPoint.Application")
```

8. When opening other applications, don't forget to make them visible:

```
appPPT.Visible = True
```

9. Compose the Open statement for PowerPoint that combines the myPath, myFileName, and myExtension variables:

```
appPPT.Presentations.Open Filename:=myPath & myFileName & myExtension
```

10. When completed, the macro looks like this, with comments that have been added to explain each step:

```
Sub OpenPowerPoint()  
    'Declare variables for path, file name and file extension.  
    Dim myPath As String, myFileName As String, myExtension As String  
    'Define the myPath and myFileName variables.  
    myPath = "C:\Your\file\Path\"  
    myFileName = "PowerPointExample1"
```

```
'Using an If structure and depending on the host computer's Office version,  
'define the extension of the PowerPoint file.
```

```
If Val(Application.Version) = 11 Then  
    myExtension = ".ppt"  
Else
```

```
myExtension = ".pptx"
End If

'Declare a variable for what will be the PowerPoint object.
'Set the object to late binding by using the CreateObject method.
Dim appPPT As Object
Set appPPT = CreateObject("PowerPoint.Application")

'Make sure you include the command to make the application visible.
appPPT.Visible = True
'Open the PowerPoint file.
appPPT.Presentations.Open Filename:=myPath & myFileName & myExtension
End Sub
```

11. Press Alt+Q to return to the worksheet. Press Alt+F8 to show the Macro dialog box, and test the macro by selecting the macro name and clicking the Run button.

REFERENCE Please select the video for Lesson 29 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

30

Working with Word from Excel

With the ubiquitous presence of Microsoft Office, a common task is to create and maintain documents in Microsoft Word that either accompany, or include as part of their narrative content, data and information from Excel workbooks. From your own experience, you have probably seen situations that call for information from Word documents to be appended, printed, or exported from Word into your Excel workbook.

Word and Excel work very well together in sharing data across their respective applications. You can automate these tasks with VBA macros right from Excel, to provide your workbook projects with robust and user-friendly methods of integrating data with Word.

ACTIVATING A WORD DOCUMENT

In Lesson 29, you saw a macro named `LateBindingTest` that opened a Word document named `myWordDoc.docx`. However, in this complicated world of ours, a seemingly simple task like activating a Word document involves a few considerations:

- Word might not be open.
- Word is open but the document itself is not open.
- The Word document is already open.
- The Word document you want to open does not exist.

For such tasks that have multiple considerations, the “divide and conquer” approach is a good way to cover your bases. If you take each consideration in turn, you can craft a single macro to handle the entire process seamlessly.

Activating the Word Application

The basic premise of activating Word is that you must tell Excel you are leaving Excel altogether for a totally different application destination. The `GetObject` function is a reliable way to do this, as shown in the following macro:

```
Sub ActivateWord()
Dim wdApp As Object
Set wdApp = GetObject(, "Word.Application")
wdApp.Activate
End Sub
```

The `GetObject` function has two arguments, the first of which is an optional pathname argument that tells VBA where to look for a specified object. Because the pathname is not specified (which it need not be because it is optional), `GetObject` activates Word, because `Word.Application` is the object being specified in the second argument.

But what if Word is not open? If you try running the `ActivateWord` macro without Word being open, a runtime error occurs because VBA is being told to activate an object that can't be activated. You need to insert an error bypass in your macro to tell VBA to activate Word only if Word is open, and to open and then activate Word only if Word is closed.

You can accomplish this with the `On Error Resume Next` statement that monitors runtime error number 429, which is the VBA error number that occurs with the `GetObject` function if Word is not open. In that case, VBA opens a new instance of Word, as shown in the following modified `ActivateWord` macro:

```
Sub ActivateWord()
Dim wdApp As Object
On Error Resume Next
Set wdApp = GetObject(, "Word.Application")
If Err.Number = 429 Then
    Err.Clear
    Set wdApp = CreateObject("Word.Application")
    wdApp.Visible = True
End If
wdApp.Activate
End Sub
```

Opening and Activating a Word Document

Now that you have Word open, it's reasonable to assume that the next item on your agenda is to open an existing Word document or to create a new Word document. If the plan is to open an existing document, a wise programming practice is to account for the possibility that the document does not exist in the specified folder path.

NOTE *You never know—files get deleted, have their names changed, or get moved from one folder to another. A VBA runtime error will eventually come back to bite you when a command is given to open a file that has an unrecognized name or location.*

For demonstration purposes, say you maintain a Word document named `myWordDoc.docx` in the folder path `C:\Your\File\Path\`. Before you attempt to open the document, check the directory to make sure it resides in the expected folder path. If the Word document is not where your macro

thinks it should be, exit the macro with a message box informing the user why the process could not be completed.

Finally, your macro needs to keep its eyes on the prize (that being the Word document), which might already be open if Word is already open. You can see there's a lot to remember, but this is what macros are for... tell them once and they'll do what they're told. Here is the complete modification of the ActivateWord macro that wraps it all up into a single package:

```
Sub ActivateWord()
    'Declare Object variables for the Word application and document.
    Dim WdApp As Object, wddoc As Object
    'Declare a String variable for the example document's name and folder path.
    Dim strDocName As String

    'On Error statement if Word is not already open.
    On Error Resume Next
    'Activate Word if it is already open.
    Set WdApp = GetObject(, "Word.Application")
    If Err.Number = 429 Then
        Err.Clear
        'Create a Word application if Word is not already open.
        Set WdApp = CreateObject("Word.Application")
    End If

    'Make sure the Word application is visible.
    WdApp.Visible = True

    'Define the strDocName String variable.
    strDocName = "C:\Your\Path\myWordDoc.docx"
    'Check the directory for the presence of the document
    'name in the folder path.
    'If it is not recognized, inform the user and exit the macro.
    If Dir(strDocName) = "" Then
        MsgBox "The file myWordDoc.docx" & vbCrLf & _
            "was not found in the folder path" & vbCrLf & _
            "C:\Your\Path\", _
            vbExclamation, _
            "Sorry, that document name does not exist."
        Exit Sub
    End If

    'Activate the Word application.
    WdApp.Activate
    'Set the Object variable for the Word document's full name and folder path.
    Set wddoc = WdApp.Documents(strDocName)
    'If the Word document is not already open, then open it.
    If wddoc Is Nothing Then Set wddoc = WdApp.Documents.Open(strDocName)
    'The document is open, so activate it.
    wddoc.Activate

    'Release system memory that was reserved for the two Object variables.
    Set wddoc = Nothing
    Set WdApp = Nothing
End Sub
```

CREATING A NEW WORD DOCUMENT

You can easily create a new Word document from scratch with the `Documents.Add` method statement associated with your Word application `Object` variable. For example, in the previous macro named `ActivateWord`, the Word application was declared as `Dim WdApp As Object`. Toward the end of that macro, before the `wddoc` and `WdApp` Object variables were set to `Nothing`, you could insert this line to add a new document to that open instance of Word:

```
WdApp.Documents.Add
```

You'll typically create a new Word document for the purpose of holding some kind of narrative or data, which means you want to save your new document. Tapping into many of the same processes that were covered in the `ActivateWord` macro, here is an example of a macro that creates and saves a new Word document:

```
Sub CreateWordDoc()

    'Declare Object variables for the Word application and new document.
    Dim objWordApp As Object, objWordDoc As Object

    'On Error statement if Word is not already open.
    On Error Resume Next
    'Activate Word if it is already open.
    Set objWordApp = GetObject(, "Word.Application")
    If Err.Number = 429 Then
        Err.Clear
        'Create a Word application if Word is not already open.
        Set objWordApp = CreateObject("Word.Application")
    End If

    'Make sure the Word application is visible.
    objWordApp.Visible = True

    'Activate the Word application.
    objWordApp.Activate

    'Create your new Word document.
    Set objWordDoc = objWordApp.Documents.Add
    'Save your new Word document in a folder path.
    objWordDoc.SaveAs "C:\Your\File\Path\myNewWordDoc.docx"

    'Release system memory that was reserved for the two Object variables.
    Set objWordApp = Nothing
    Set objWordDoc = Nothing

End Sub
```

COPYING AN EXCEL RANGE TO A WORD DOCUMENT

Suppose you have a table of data in your Excel workbook on Sheet1 in range A1:H25. You want to export the table into an existing Word document named `myWordDoc.docx`, which you know exists and you know is closed. To make it interesting, say the task calls for the following set of actions:

1. Open Word.
2. Open myWordDoc.docx.
3. Export the data table from Excel into the myWordDoc.docx document.
4. Save myWordDoc.docx.
5. Close myWordDoc.docx.

The following macro accomplishes this task very quickly. Note that you can copy a worksheet's used range or current region of a cell; you do not need to refer to a specific range address as this example does:

```
Sub ExportFromExcelToWord()
    'Turn off ScreenUpdating.
    Application.ScreenUpdating = False

    'Copy the Excel range to be exported.
    Worksheets("Sheet1").Range("A1:H25").Copy

    'Declare object variables.
    Dim WdApp As Object, wddoc As Object

    'Open Word
    Set WdApp = CreateObject("Word.Application")
    'Open the Word document that will accept the exported data.
    Set wddoc = WdApp.Documents.Open(Filename:="C:\Your\File\Path\myWordDoc.docx")
    'Paste the copied data from Excel to the Word document.
    wddoc.Range.Paste
    'Close the Word document and save changes.
    wddoc.Close savechanges:=True
    'Quit the Word application.
    WdApp.Quit

    'Set the Object variables to Nothing to release system memory.
    Set wddoc = Nothing
    Set WdApp = Nothing

    'Exit Copy mode.
    Application.CutCopyMode = False
    'Turn ScreenUpdating back on.
    Application.ScreenUpdating = True
End Sub
```

PRINTING A WORD DOCUMENT FROM EXCEL

To print a Word document, you can use the PrintOut method to print the entire document, or only a portion of the document if you so choose. The following macro shows an example of opening and printing a Word document:

```
Sub PrintWordDoc()
    'Declare object variables.
```

```

Dim WdApp As Object, wddoc As Object

'Open Word
Set WdApp = CreateObject("Word.Application")
'Open the Word document to be printed.
Set wddoc = WdApp.Documents.Open(Filename:="C:\Your\File\Path\myWordDoc.docx")
'Print the entire Word document.
WdApp.ActiveDocument.PrintOut
'Give the print job 5 seconds to complete before closing Word.
Application.Wait Now + TimeSerial(0, 0, 5)
'Close the Word document, no need to save changes.
wddoc.Close savechanges:=False
'Quit the Word application.
WdApp.Quit

'Set the Object variables to Nothing to release system memory.
Set wddoc = Nothing
Set WdApp = Nothing
End Sub

```

NOTE You might have noticed that this macro—and a couple of others in this lesson—do not include the statement to make the Word application visible. It's easy to forget that you have an open application if you cannot see it. The point is, remember to include the Close and Quit statements in your macros when opening applications and files that you intend to be closed when the macro is completed. Otherwise, when you rerun the macro, you get read-only messages and error messages, because VBA interprets your coded instructions as an attempt to re-open a file that is already open.

If you want to print only a portion of the Word document, for example only page 2, then in the preceding PrintWordDoc macro, substitute the statement

```
WdApp.ActiveDocument.PrintOut
```

with

```
WdApp.ActiveDocument.PrintOut pages:="2"
```

IMPORTING A WORD DOCUMENT TO EXCEL

There may be times when you want to import some text from Word into Excel. Admittedly this is not a common task, because Excel cells are not meant to serve as word processing instruments for extensive amounts of text. But because it's possible, here's a macro that opens a Word document, copies the second paragraph, and pastes that text into the active cell of your workbook:

```

Sub ImportToExcelFromWord()
'Declare Object variables for the Word application and document.
Dim WdApp As Object, wddoc As Object

```

```
'Declare a String variable for the example document's name and folder path.  
Dim strDocName As String  
  
'On Error statement if Word is not already open.  
On Error Resume Next  
'Activate Word if it is already open.  
Set WdApp = GetObject(, "Word.Application")  
If Err.Number = 429 Then  
    Err.Clear  
'Create a Word application if Word is not already open.  
Set WdApp = CreateObject("Word.Application")  
End If  
  
'Make sure the Word application is visible.  
WdApp.Visible = True  
  
'Define the strDocName String variable.  
strDocName = "C:\Your\Path\myWordDoc.docx"  
  
'Activate the Word application.  
WdApp.Activate  
  
'Set the Object variable for the Word document's full name and folder path.  
Set wddoc = WdApp.Documents(strDocName)  
'If the Word document is not already open, then open it.  
If wddoc Is Nothing Then Set wddoc = WdApp.Documents.Open(strDocName)  
  
'The document is open, so activate it.  
wddoc.Activate  
  
'Copy paragraph 2  
wddoc.Paragraphs(2).Range.Copy  
'Activate your workbook and paste the copied text into the active cell.  
ThisWorkbook.Activate  
'Paste paragraph 2 from the Word document.  
ActiveSheet.Paste  
'Close the Word document, no need to save changes.  
wddoc.Close Savechanges:=False  
'Quit the Word application.  
WdApp.Quit  
  
'Release the system memory that was reserved for the two Object variables.  
Set wddoc = Nothing  
Set WdApp = Nothing  
End Sub
```

TRY IT

For this lesson, you write a macro that uses an `InputBox` to ask for the name of a Word document to be opened from a predetermined folder. If the Word document exists, it is opened, but if it does not exist, the user is advised of that.

Lesson Requirements

To get the sample workbook you can download Lesson 30 from the book's website at www.wrox.com/excelvba24hour.

Step-by-Step

1. From any worksheet in your Excel workbook, press Alt+F11 to go to the Visual Basic Editor.
2. From the VBE menu bar, click Insert \Rightarrow Module.
3. In the module you just created, type `Sub OpenRequestedWordDoc` and press Enter. VBA automatically places a pair of empty parentheses at the end of the `Sub` line, followed by an empty line, and the `End Sub` line below that. Your macro looks like this so far:

```
Sub OpenRequestedWordDoc()  
  
End Sub
```

4. Declare a `String` type variable for the predetermined folder path:

```
Dim myPath As String
```

5. Define the `String` type variable for the example folder path:

```
myPath = "C:\Your\File\Path\"
```

6. Declare a `String` type variable for the anticipated `InputBox` entry:

```
Dim myFileName As String
```

7. Define the `myFileName` variable with an `InputBox` to ask the user for the name of the Word document to be opened from the predetermined folder path. Note the opportunity to use the `InputBox`'s optional third argument to show an example entry of a document's full name including its extension:

```
myFileName = InputBox  
("Enter the full Word document name to be opened" & Chr(10) &  
"from the folder path " & myPath & ":",  
"What file name with extension do you wish to open?",  
"YourDocumentName.docx")
```

8. Exit the macro if nothing is entered or if the Cancel button is clicked:

```
If myFileName = "" Then Exit Sub
```

9. Declare a `String` type variable for the combined folder path and document name:

```
Dim myDocName As String
```

10. Define the `String` type variable for the combined folder path and document name:

```
myDocName = myPath & myFileName
```

11. Check to see whether the Word document name exists in the folder path. If it does not, advise the user and exit the macro. Notice that you are providing a piece of user-friendly information in the message box that reminds the user that the document name entered was not found:

```
If Dir(myDocName) = "" Then
    MsgBox "The file " & myFileName & vbCrLf &
    "was not found in the folder path" & vbCrLf &
    myPath & ".", _
    vbExclamation, _
    "No such animal."
    Exit Sub
End If
```

12. At this point, the Word document would be determined to exist in the folder. Declare `Object` variables for the Word application and the Word document:

```
Dim appWord As Object, wdDoc As Object
```

13. Using late binding, create a Word application:

```
Set appWord = CreateObject("Word.Application")
```

14. Make the created Word application visible:

```
appWord.Visible = True
```

15. Open the requested Word document name using the `Set` statement for your `wdDoc` variable:

```
Set wdDoc = appWord.Documents.Open(myDocName)
```

16. Release the reserved memory in VBA for the declared `Object` type variables now that they have served their purpose and are no longer needed:

```
Set wdDoc = Nothing
Set appWord = Nothing
```

17. Go ahead and test your macro, which looks like this in its entirety:

```
Sub OpenRequestedWordDoc()
    'Declare a String variable for the predetermined folder path.
    Dim myPath As String
    'Define the String variable with the example folder path.
    myPath = "C:\Your\File\Path\"

    'Declare a String variable for the anticipated InputBox entry.
    Dim myFileName As String

    'Show the InputBox to ask the user for the name of the Word
    'document they want to open from the predetermined folder path.
    myFileName = InputBox(
        "Enter the full Word document name to be opened" & Chr(10) &
```

```
"from the folder path " & myPath & ":", _  
"What file name with extension do you wish to open?", _  
"YourDocumentName.docx")  
  
'Exit the macro if nothing is entered or the Cancel button is clicked.  
If myFileName = "" Then Exit Sub  
  
'Declare a String variable for the combined folder path  
'and document name.  
Dim myDocName As String  
'Define the String variable for the combined folder path  
'and document name.  
myDocName = myPath & myFileName  
  
'Check to see if the Word document name actually exists  
'in the folder path.  
'If it does not, then advise the user and exit the macro.  
If Dir(myDocName) = "" Then  
MsgBox "The file " & myFileName & vbCrLf & _  
"was not found in the folder path" & vbCrLf & _  
myPath & ".", _  
vbExclamation, _  
"No such animal."  
Exit Sub  
End If  
  
'At this point, the Word document is determined to exist  
'in the folder.  
'Declare Object variables for the Word application and  
'the Word document.  
Dim appWord As Object, wdDoc As Object  
  
'Using late binding in this example, create a Word application.  
Set appWord = CreateObject("Word.Application")  
'Make the created Word application visible.  
appWord.Visible = True  
'Open the requested Word document name.  
Set wdDoc = appWord.Documents.Open(myDocName)  
  
'Release the reserved memory in VBA for the declared Object variables  
'now that they have served their purpose and are no longer needed.  
Set wdDoc = Nothing  
Set appWord = Nothing  
  
End Sub
```

REFERENCE Please select the video for Lesson 30 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

31

Working with Outlook from Excel

Microsoft Outlook is the e-mail client application that is included in Microsoft's Office suite. In addition to e-mail management, Outlook also provides personal information management capabilities with its Calendar, Contacts, and Task Manager features. Each of these components in Outlook can be controlled from Excel with VBA.

NOTE *With all the competing e-mail clients to choose from, Outlook continues to be far and away the world's most popular e-mail application. Chances are pretty good that Outlook is your e-mail client at work or at home, or it is being used by the recipients of e-mails you send.*

OPENING OUTLOOK

Before diving into the programming of Outlook from Excel, it's worth noting that Outlook is different than Excel, Word, Access, and PowerPoint in one key respect. Unlike those other Office applications for which you might create multiple instances in your work, Outlook is not the kind of application for doing that. When it comes to handling e-mails, tasks, and calendars, it's just common sense to have only a single instance of Outlook open at any one time.

The following macro first checks to see if Outlook is already open, and if so, allows your macro to continue. If Outlook happens to be closed, the macro terminates, with a message box alerting you to please open Outlook in order to continue:

```
Sub OpenOutlook()
    Dim objOutlook As Object
    On Error Resume Next
    Set objOutlook = GetObject(, "Outlook.Application")
    If objOutlook Is Nothing Then
        Err.Clear
    End If
End Sub
```

```

MsgBox "Cannot continue, Outlook is not open.", , _
"Please open Outlook and try again."
Exit Sub
Else
MsgBox "The rest of your code goes here.", , "Outlook is open!"
End If
End Sub

```

NOTE You might have noticed that my OpenOutlook macro does not follow the same process of opening Outlook if it is not already open, such as I have shown how to do for Word and PowerPoint. The reason is that Outlook is a different animal than other Office applications with regard to how it opens using late binding. The programming code with late binding to open Outlook, without it already being open, involves VBA methods beyond the scope of this introductory lesson. That's OK, because in everyday practice, it's not uncommon to show user-friendly messages, such as the OpenOutlook macro does, to request a simple manual step be performed before proceeding.

COMPOSING AN E-MAIL IN OUTLOOK FROM EXCEL

Most of the time, when you open Outlook, whether manually or with VBA, it's for the purpose of doing something, such as to receive or send e-mails, but also to update your calendar or manage your task list. Building upon the previous code that opens or activates Outlook, this section explains how to compose and send a complete e-mail message from Excel.

Creating a MailItem Object

Where VBA is concerned, `MailItem` is an Outlook object that you know quite well as a typical e-mail message that arrives in your Inbox. The `MailItem` object is made up of the familiar fields `To`, `CC`, and `Subject`. The other components of the `MailItem` object are the `Body` where you type the text of your message, an optional level of Importance, and maybe an attachment.

When you want to compose an e-mail with VBA, you declare a variable for the `MailItem` object and set it as a created item of the `Outlook` application object. For example, the following macro creates an e-mail message with a workbook attachment. Figure 31-1 shows the e-mail that is created.

```

Sub SendEmail()
    'Declare the Object variables for Outlook.
    Dim objOutlook As Object
    'Verify Outlook is open.
    On Error Resume Next
    Set objOutlook = GetObject(, "Outlook.Application")

    'If Outlook is not open, end the Sub.
    If objOutlook Is Nothing Then

        Err.Clear

```

```

MsgBox _
"Cannot continue, Outlook is not open.", , _
"Please open Outlook and try again."
Exit Sub

'Outlook is determined to be open, so OK to proceed.
Else

'Establish an Object variable for a mailitem.
Dim objMailItem As Object
Set objMailItem = objOutlook.CreateItem(0)

'Build the mailitem.
With objMailItem
    .To = "tom@atlaspm.com"
    .CC = "tomurtis@gmail.com"
    .Subject = "Testing Lesson 31 email code"
    .Importance = 1 'Sets it as Normal importance (Low = 0 and High = 2)
    .Body = "Hello, this is a test." & vbCrLf & "Have a nice day."
    .Attachments.Add "C:\Your\File\Path\YourFileName.xlsx"
    'Change the Display command to Send without reviewing the email.
    .Display
End With

'Close the If block.
End If

End Sub

```

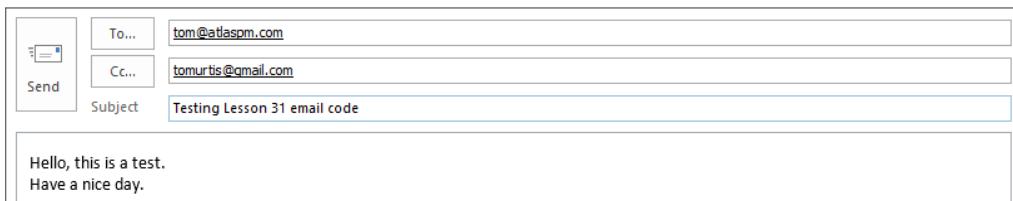


FIGURE 31-1

NOTE The `Importance` property is optional; you don't need to include it. If you do include it as I did with this example, the 1 is a reference to Normal Importance. Low Importance would be 0 and High Importance would be 2. Also, in all the e-mail examples in this lesson, `Display` is utilized rather than `Send`, so that when you test these examples, you can actually see the resulting `MailItem` object without sending it.

Transferring an Excel Range to the Body of Your E-mail

In the preceding example of composing a `MailItem` object, the body of the e-mail message was hard-coded into the macro, with this statement:

```
.Body = "Hello, this is a test." & vbCrLf & "Have a nice day."
```

You might be interested to know that you can represent a range of worksheet data in the body of an e-mail message. One way to accomplish that is to loop through the cells and create a text string, with a line break character to simulate each row item. Figure 31-2 shows a simple list that is referred to in this example.

	A	B	C
1	Name	Position	
2	Judy	Regional Manager	
3	Jennifer	General Manager	
4	Mia	Events Coordinator	
5	Ricky	Chief Engineer	
6	Alicia	District Manager	
7	Dana	General Manager	
8	Wilbur	Office Manager	
9	Josh	IT and Networks	
10	Emily	Records Manager	
11	William	Security	
12	Bob	Health and Fitness	
13	Mike	Sales	
14	James	Counsel	
15			

FIGURE 31-2

To copy the list, declare a `String` variable for the text values as you loop through each cell in the list, and declare `Long` variables for the count of rows and columns in the range you are copying. In this example, two columns are being copied. However, the range you want to copy might have an unknown number of rows and columns to be represented in your e-mail.

The following macro named `BuildDynamicString` builds a continuous string, taking into consideration a dynamic range based on the `CurrentRegion` property of cell A1 that is shown in Figure 31-2. This next macro demonstrates how to build a string that will be placed in the body of an e-mail. The message box shown in Figure 31-3 gives you a quick glance at this code's immediate result.

```
Sub BuildDynamicString()
    'Declare a String variable for the worksheet data.
    Dim strtext As String
    'Declare Long variables for the range's Row and Columns.
    Dim xRow As Long, xColumn As Long
    'Build the string that is the text inside the range
    'you want to represent in the Body of the email.
    For xRow = 1 To Range("A1").CurrentRegion.Rows.Count
        For xColumn = 1 To Range("A1").CurrentRegion.Columns.Count
            strtext = strtext & Range("A1").Cells(xRow, xColumn).Value & vbTab
        Next xColumn
        strtext = strtext & Chr(10)
    Next xRow
    'Show the string in a message box, just for demo purposes.
    MsgBox strtext, , "Example"
End Sub
```

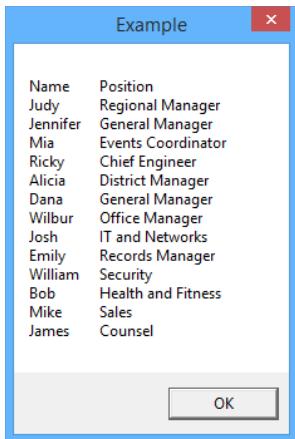


FIGURE 31-3

PUTTING IT ALL TOGETHER

The following macro ties together all the previous code examples in this lesson. Figure 31-4 shows what your e-mail would look like in Outlook 2013 after running the macro named `ExampleEmail`.

```
Sub ExampleEmail()

    'Declare the Object variables for Outlook.
    Dim objOutlook As Object
    'Verify Outlook is open.
    On Error Resume Next
    Set objOutlook = GetObject(, "Outlook.Application")

    'If Outlook is not open, end the Sub.
    If objOutlook Is Nothing Then

        Err.Clear
        MsgBox "Cannot continue -- Outlook is not open.", , _
        "Please open Outlook and try again."
        Exit Sub

    'Outlook is determined to be open, so OK to proceed.
    Else

        'Declare a String variable for the worksheet data.
        Dim strText As String
        'Declare Long variables for the range's row and columns.
        Dim xRow As Long, xColumn As Long
        'Build the string that is the text inside the range
        'you want to represent in the Body of the email.
        For xRow = 1 To Range("A1").CurrentRegion.Rows.Count
            For xColumn = 1 To Range("A1").CurrentRegion.Columns.Count
                strText = strText & Range("A1").Cells(xRow, xColumn).Value & vbTab
            Next xColumn
        Next xRow

    End If

End Sub
```

```

strtext = strtext & Chr(10)
Next xRow

'Establish an Object variable for a mailitem.
Dim objMailItem As Object
Set objMailItem = objOutlook.CreateItem(0)

'Build the mailitem and attach a workbook.
With objMailItem
    .To = "tom@atlaspm.com"
    .CC = "tomurtis@gmail.com"
    .Subject = "Testing Lesson 31 email code"
    .Importance = 1 'Sets it as Normal importance (Low = 0 and High = 2)
    .Body = "List of employees and positions:" & vbCrLf & vbCrLf & strtext
    'Change the Display command to Send without reviewing the email.
    .Display
End With

'Close the If block.
End If

'Release object variables from system memory.
Set objOutlook = Nothing
Set objMailItem = Nothing

End Sub

```

NOTE Before testing the ExampleEmail macro, you probably need to modify the folder path and filename of the attachment. If you want to test the macro without attaching a file, you can simply delete or comment out the `Attachments.Add` statement.

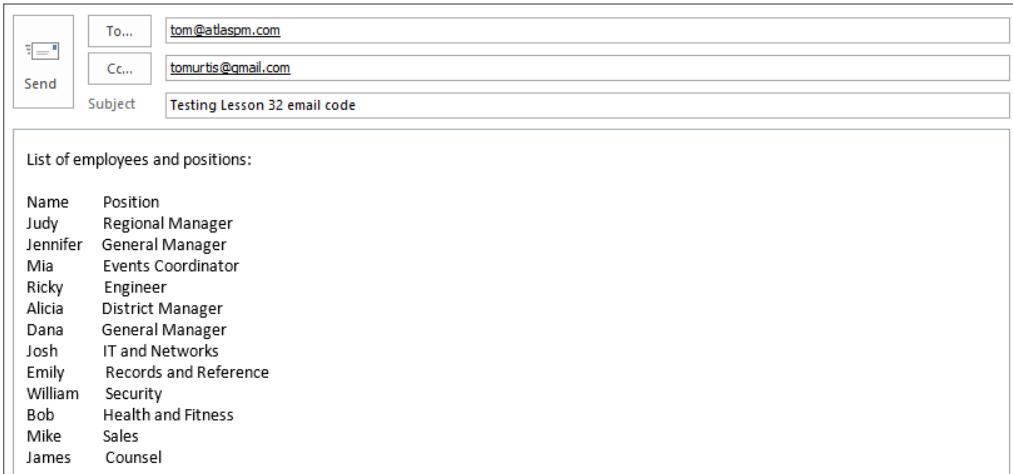


FIGURE 31-4

E-MAILING A SINGLE WORKSHEET

You can e-mail a single worksheet using `SendMail` with Microsoft Outlook. The following macro copies the active worksheet and sends it as the lone worksheet in its own workbook:

```
Sub EmailSingleSheet()
    ActiveSheet.Copy
    On Error Resume Next
    ActiveWorkbook.SendMail "tom@atlaspm.com", "Test of single sheet."
    Err.Clear
    ActiveWorkbook.Close False
End Sub
```

NOTE *SendMail can send a single worksheet as an attachment by housing that worksheet in its own workbook and e-mailing it. SendMail does not require specifying a Simple Mail Transport Protocol (SMTP) server; it sends the mail using your installed mail system. This has the advantage of bypassing much of the Outlook-related code you've seen so far, but it comes with disadvantages, such as limited ability to attach files, and no available CC argument.*

NOTE *A worksheet in Excel cannot exist on its own; a worksheet must be housed in a parent Excel workbook.*

TRY IT

In this lesson, you write a macro in Excel that creates an e-mail in Microsoft Outlook for multiple recipients whose addresses are listed in column A of your worksheet. The macro populates the e-mail's To field with the recipients' names, and attaches the active Excel workbook to that e-mail.

Lesson Requirements

To get the sample workbook, you can download Lesson 31 from the book's website at www.wrox.com/excelvba24hour.

Step-by-Step

1. In column A of your worksheet, list a few sample recipient names. For example:
 - In cell A1 enter `no_one@nowhere.com`.
 - In cell A2 enter `anyone@anywhere.com`.
 - In cell A3 enter `someone@somewhere.com`.

2. Press Alt+F11 to go to the Visual Basic Editor.
3. From the menu bar at the top of the VBE, click Insert \Rightarrow Module.
4. In the module you just created, type **Sub EmailAttachmentRecipients** and press Enter. VBA automatically places a pair of empty parentheses at the end of the Sub line, followed by an empty line, and the End Sub line below that. Your macro looks like this so far:

```
Sub EmailAttachmentRecipients ()  
End Sub
```

5. Establish the Object variable for Outlook:

```
Dim objOutlook As Object
```

6. Verify whether Outlook is open:

```
On Error Resume Next  
Set objOutlook = GetObject(, "Outlook.Application")
```

7. If Outlook is not open, end the macro:

```
If objOutlook Is Nothing Then  
Err.Clear  
MsgBox "Cannot continue -- Outlook is not open.", , _  
"Please open Outlook and try again."  
Exit Sub
```

8. At this point, Outlook is verified to be open. Establish an Object variable for a mailitem:

```
Else  
Dim objMailItem As Object  
Set objMailItem = objOutlook.CreateItem(0)
```

9. Declare a String variable for the recipient list, and a Long variable for the count of cells in column A that contain e-mail addresses:

```
Dim strTo As String  
Dim i As Integer  
strTo = ""  
i = 1
```

10. Loop through the recipient e-mail addresses you entered from Step 1 to build a continuous string where each recipient address is separated by a semicolon and a space, just as it would appear in an Outlook To field:

```
Do  
strTo = strTo & Cells(i, 1).Value & "; "  
i = i + 1  
Loop Until IsEmpty(Cells(i, 1))  
'Remove the last two characters from the string,  
'which are an unneeded semicolon and space.  
strTo = Mid(strTo, 1, Len(strTo) - 2)
```

11. Display the e-mail message, including the attachment of the active workbook:

```
With objMailItem
    .To = strTo
    .Subject = "Test of multiple recipients"
    .Body = _
        "Hello everyone, testing multiple recipients with a workbook attachment."
    .Attachments.Add ActiveWorkbook.FullName
    .Display 'Change to Send
End With
```

NOTE *The active workbook you are attaching must be an actual workbook that has been named and saved, or the code line .Attachments.Add ActiveWorkbook.FullName will fail.*

12. Release Object variables from system memory:

```
Set objOutlook = Nothing
Set objMailItem = Nothing
```

13. Close the If structure:

```
End If
```

14. When your macro is complete, it should look like this:

```
Sub EmailAttachmentRecipients()

    'Declare the Object variable for Outlook.
    Dim objOutlook As Object
    'Verify Outlook is open.
    On Error Resume Next
    Set objOutlook = GetObject(, "Outlook.Application")

    'If Outlook is not open, end the Sub.
    If objOutlook Is Nothing Then
        Err.Clear
        MsgBox "Cannot continue -- Outlook is not open.", , _
            "Please open Outlook and try again."
        Exit Sub

    'Outlook is determined to be open, so OK to proceed.
    Else

        'Establish an Object variable for a mailitem.
        Dim objMailItem As Object
        Set objMailItem = objOutlook.CreateItem(0)

        'Declare a String variable for the recipient list,
        'and a Long variable for the count of cells in column A
        'that contain email addresses.
```

```
Dim strTo As String
Dim i As Integer
strTo = ""
i = 1

'Loop through the recipient email addresses you entered from Step 1,
'in order to build a continuous string where each recipient address
'is separated by a semicolon and a space, just as it would appear
'in an Outlook "To" field.
Do
    strTo = strTo & Cells(i, 1).Value & "; "
    i = i + 1
Loop Until IsEmpty(Cells(i, 1))
'Remove the last two characters from the string,
'which are an unneeded semicolon and space.
strTo = Mid(strTo, 1, Len(strTo) - 2)

'Display the email message, including the attachment of the active workbook.
With objMailItem
    .To = strTo
    .Subject = "Test of multiple recipients"
    .Body = _
        "Hello everyone, testing multiple recipients with a workbook attachment."
    .Attachments.Add ActiveWorkbook.FullName
    .Display 'Change to Send
End With

'Release object variables from system memory.
Set objOutlook = Nothing
Set objMailItem = Nothing

'Close the If structure.
End If

End Sub
```

15. Press Alt+Q to return to the worksheet. Press Alt+F8 to show the Macro dialog box, and test the macro by selecting its name and clicking the Run button.

REFERENCE Please select the video for Lesson 31 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

32

Working with Access from Excel

As terrific a product as Excel is, there can come a point when the volume of data you are working with will exceed Excel's capacity for storing records. Even with more than one million available rows starting with version 2007, some projects require a larger data management platform with Microsoft Access. If you plan to develop projects for business clients, sooner or later you'll encounter a client that uses Access for its relational database capabilities.

Using Excel VBA with the storage capabilities of an Access relational database is a powerful combination for front-end data management. This lesson offers examples for adding, retrieving, and updating records in Access data tables from the familiar comfort of your Excel workbook.

ADDING A RECORD TO AN ACCESS TABLE

Among the more common actions you'll do when interacting with Access from Excel is to transfer records from an Excel worksheet to an Access database table, and vice versa. Suppose there is an Access database named `Database2.accdb` that contains a table named `Table1` with six fields. In Sheet5 of your Excel workbook, you amass records during the day that are added to `Table1` at the end of the workday.

NOTE *A reference to the Microsoft ActiveX Data Objects 2.8 Library is required for the code in this lesson to run. Before attempting to run the macros, get into the VBE and from the menu bar, click Tools → References. Navigate to the reference for Microsoft ActiveX Data Objects 2.8 Library (or the highest Data Objects Library number you see starting with the number 2), select it as indicated in Figure 32-1, and click OK.*

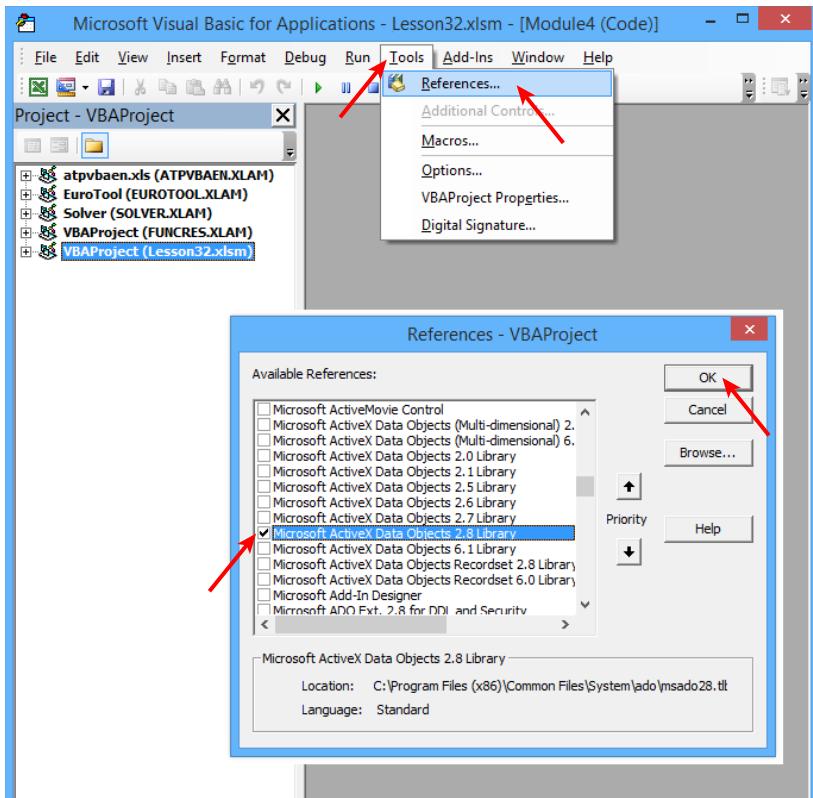
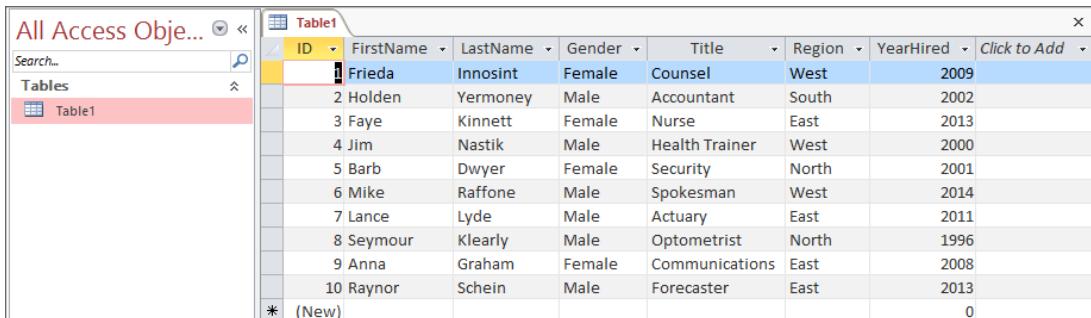


FIGURE 32-1

Suppose you have a table named `Table1` in an Access database file to maintain a list of employees at your company, as shown in Figure 32-2. As the company hires new employees, the information is first recorded in an Excel workbook for other internal business purposes. Figure 32-3 shows that two new employees were hired in 2015, whose information needs to be appended to the existing table in Access.

To automate the task of updating a table in Access with records from Excel, you would maintain the Excel table with the fields in the same order, and the field headers spelled exactly the same way as they are found in the Access table you will update. Notice that the field headers are identical in Figures 32-2 and 32-3.

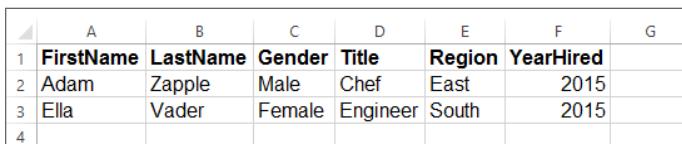
NOTE In the previous paragraph, I wrote that field headers on your spreadsheet and in your Access table must be identical, in the same order and spelled exactly the same. I'm using this tip as a reminder that any difference, such as a stray spacebar character or misspelling, will cause the following macro to fail. If you get a runtime error number 3265, your first move should be to check for any differences in how your field headers are arranged and/or spelled.



The screenshot shows the Microsoft Access 'All Access Objects' window. On the left, under 'Tables', 'Table1' is selected. To its right is a grid view of the 'Table1' data. The columns are labeled: ID, FirstName, LastName, Gender, Title, Region, YearHired, and Click to Add. The data includes 10 rows of employee information, with row 11 labeled '(New)'.

ID	FirstName	LastName	Gender	Title	Region	YearHired	Click to Add
1	Frieda	Innosint	Female	Counsel	West	2009	
2	Holden	Yermoney	Male	Accountant	South	2002	
3	Faye	Kinnett	Female	Nurse	East	2013	
4	Jim	Nastik	Male	Health Trainer	West	2000	
5	Barb	Dwyer	Female	Security	North	2001	
6	Mike	Raffone	Male	Spokesman	West	2014	
7	Lance	Lyde	Male	Actuary	East	2011	
8	Seymour	Klearly	Male	Optometrist	North	1996	
9	Anna	Graham	Female	Communications	East	2008	
10	Raynor	Schein	Male	Forecaster	East	2013	
*	(New)					0	

FIGURE 32-2



The screenshot shows an Excel spreadsheet with columns labeled A through G. Row 1 contains column headers: FirstName, LastName, Gender, Title, Region, and YearHired. Rows 2 and 3 contain data for Adam and Ella respectively. Row 4 is blank.

	A	B	C	D	E	F	G
1	FirstName	LastName	Gender	Title	Region	YearHired	
2	Adam	Zapple	Male	Chef	East	2015	
3	Ella	Vader	Female	Engineer	South	2015	
4							

FIGURE 32-3

The following Excel macro named AppendRecords appends the two new employee records from the Excel worksheet into Table1 of the Database2 file. Figure 32-4 shows how Table1 looks after the AppendRecords macro adds the two new employee records.

```

Sub AppendRecords()
    'Declare variables.
    Dim dbConnection As ADODB.Connection
    Dim dbFileName As String
    Dim dbRecordset As ADODB.Recordset
    Dim xRow As Long, xColumn As Long
    Dim LastRow As Long

    'Go to the worksheet containing the records you want to transfer.
    Worksheets("Sheet5").Activate
    'Determine the last row of data based on column A.
    LastRow = Cells(Rows.Count, 1).End(xlUp).Row

    'Create the connection to the database.
    Set dbConnection = New ADODB.Connection
    'Define the database file name.
    dbFileName = "C:\Your\File\Path\Database2.accdb"

    'Define the Provider and open the connection.
    With dbConnection
        .Provider = "Microsoft.ACE.OLEDB.12.0;Data Source=" & dbFileName & _
        ";Persist Security Info=False;"
        .Open dbFileName
    End With
End Sub

```

```
End With
```

```
'Create the recordset.
Set dbRecordset = New ADODB.Recordset
dbRecordset.CursorLocation = adUseServer
dbRecordset.Open Source:="Table1", _
ActiveConnection:=dbConnection, _
CursorType:=adOpenDynamic, _
LockType:=adLockOptimistic, _
Options:=adCmdTable

'Load the records from Excel to Access, by looping through the rows and columns.
'Assume row 1 is the header row, so start at row 2.
For xRow = 2 To LastRow
dbRecordset.AddNew
'Assume this is an 6-column (field) table starting with column A.
For xColumn = 1 To 6
dbRecordset(Cells(1, xColumn).Value) = Cells(xRow, xColumn).Value
```

NOTE The preceding line of code will fail and result in a runtime error if any field in your Excel table contains data that is in conflict with the specified data type of its corresponding field in the Access table. For example, if the second field in your Access table is specified to be a Number data type, and in your Excel worksheet, column B has a text value in it, the macro will fail at this point because the code is attempting to place a text value into an Access field meant to accept only numbers.

```
Next xColumn
dbRecordset.Update
Next xRow

'Close the connections.
dbRecordset.Close
dbConnection.Close

'Release Object variable memory.
Set dbRecordset = Nothing
Set dbConnection = Nothing

'Alert the user that the process is complete.
MsgBox "Transfer complete!", , "Done!"

End Sub
```

	ID	FirstName	LastName	Gender	Title	Region	YearHired	Click to Add
*	1	Frieda	Innosint	Female	Counsel	West	2009	
	2	Holden	Yermoney	Male	Accountant	South	2002	
	3	Faye	Kinnett	Female	Nurse	East	2013	
	4	Jim	Nastik	Male	Health Trainer	West	2000	
	5	Barb	Dwyer	Female	Security	North	2001	
	6	Mike	Raffone	Male	Spokesman	West	2014	
	7	Lance	Lyde	Male	Actuary	East	2011	
	8	Seymour	Klearly	Male	Optometrist	North	1996	
	9	Anna	Graham	Female	Communications	East	2008	
	10	Raynor	Schein	Male	Forecaster	East	2013	
	11	Adam	Zapple	Male	Chef	East	2015	
	12	Ella	Vader	Female	Engineer	South	2015	
	*	(New)					0	

FIGURE 32-4

NOTE You are probably aware that beginning with the release of Office 97, extensions changed for Microsoft applications. For example, Excel workbooks that had the extension .xls now are either .xlsx or .xlsm. Access extensions also changed, from .mdb to .accdb, as shown in the preceding macro.

Take note of the version(s) of Excel and Access when the time comes to implement this code. Especially, the Provider line in the code is

```
.Provider = "Microsoft.ACE.OLEDB.12.0;Data Source=" _  
& dbFileName & ";Persist Security Info=False;".
```

Had this been a version of Office prior to 2007, that same line might have been

```
.Provider = "Microsoft.Jet.OLEDB.4.0"
```

or

```
.Provider = "Microsoft.Jet.OLEDB.4.0;" & "Data Source=" _  
& dbFileName & ";" & "Extended Properties=Excel 8.0;".
```

EXPORTING AN ACCESS TABLE TO AN EXCEL SPREADSHEET

As mentioned earlier, you will commonly need to import a table from an Access database into an Excel worksheet to take advantage of Excel's versatile formatting and data manipulation capabilities. To export the database's Table1 data, you define the recordset while passing an SQL string to the connection. In this example, the entire count of records in Table1 is copied to Sheet2 in your Excel workbook:

```
Sub AccessToExcel()  
  
    'Declare variables.  
    Dim dbConnection As ADODB.Connection
```

WHAT IF YOU ONLY WANT TO OPEN AN ACCESS DATABASE FILE FROM EXCEL?

A common theme you'll notice with the examples in this lesson is that Excel is acting upon the Access files by connecting to them, rather than by opening and closing them as you see in the lessons for working with Word, Outlook, and PowerPoint. You will rarely need Excel to open an Access database just for the sake of opening it.

If the situation arises where you do need to open an Access database from Excel, the following example is what I use. It works by incorporating a `ShellExecute` command in conjunction with the declaration of the `ShellExecute` function from the Windows API. The `ShellExecute` function in the Windows API performs an operation on a specified file. In this case, the specified file is the one you want to open (named `Database1.accdb` in the hypothetical directory path `C:\Your\File\Path`), and the operation is to open that database file, using the parameters in the declaration statement. This code is placed in a standard Excel VBA module just as any macro would be, and works with Windows versions from XP through Windows 8.1. If you are running Excel with the 64-bit version of Office, there is an explanation of the `PtrSafe` keyword in Lesson 28 in the section "Using the Windows API with VBA."

```
#If VBA7 Then

    Public Declare PtrSafe Function _
        ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" ( _
            ByVal hwnd As LongPtr, ByVal lpOperation As String, _
            ByVal lpFile As String, ByVal lpParameters As String, _
            ByVal lpDirectory As String, ByVal nShowCmd As Long) As LongPtr

    #Else

        Public Declare Function _
            ShellExecute Lib "shell32.dll" Alias "ShellExecuteA" ( _
                ByVal hwnd As Long, ByVal lpOperation As String, _
                ByVal lpFile As String, ByVal lpParameters As String, _
                ByVal lpDirectory As String, ByVal nShowCmd As Long) As Long

    #End If

    Sub OpenAccessDB()
        Call ShellExecute(0, "Open", "Database1.accdb", "", _
            "C:\Your\File\Path", 1)
    End Sub
```

```
Dim dbRecordset As ADODB.Recordset
Dim dbFileName As String
Dim strSQL As String
Dim DestinationSheet As Worksheet

' Set the assignments to the Object variables.
```

```
Set dbConnection = New ADODB.Connection
Set dbRecordset = New ADODB.Recordset
Set DestinationSheet = Worksheets("Sheet2")

'Define the Access database path and name.
dbFileName = "C:\Your\file\Path\Database2.accdb"
'Define the Provider for post-2007 database files.
dbConnection.Provider = "Microsoft.ACE.OLEDB.12.0;Data Source=" _
& dbFileName & ";Persist Security Info=False;"

'Use SQL's SELECT and FROM statements for importing Table1.
strSQL = "SELECT Table1.* FROM Table1;"

'Clear the destination worksheet.
DestinationSheet.Cells.Clear

With dbConnection
'Open the connection.
.Open
'The purpose of this line is to disconnect the recordset.
.CursorLocation = adUseClient
End With

With dbRecordset
'Create the recordset.
.Open strSQL, dbConnection
'Disconnect the recordset.
Set .ActiveConnection = Nothing
End With

'Copy the table1 recordset to Sheet2 starting in cell A2.
'Row 1 contains headers that will be populated at the next step.
DestinationSheet.Range("A2").CopyFromRecordset dbRecordset

'Reinstate field headers (assumes a 6-column table).
'Note that the ID field will also transfer into column A,
'so you can optionally delete column A.
DestinationSheet.Range("A1:G1").Value =
Array("ID", "FirstName", "LastName", "Gender", "Title", "Region", "YearHired")

'Close the recordset.
dbRecordset.Close
'Close the connection.
dbConnection.Close

'Release Object variable memory.
Set dbRecordset = Nothing
Set dbConnection = Nothing
Set DestinationSheet = Nothing

End Sub
```

NOTE Here's a tip to import a database table into your spreadsheet manually. It'll come with the alternating shaded rows and field header drop-down arrows, but it's fast and easy! With the database file closed, the keyboard shortcut Alt+D+D+D shows the Select Data Source window. Navigate to your database file, select its name, and click Open. In the Select Table dialog box, select the name of the table you want to import and click OK. In the Import Data dialog box, select the option for Table. Finally, select the option for Existing Worksheet and specify the cell address, or the option for New Worksheet, and click OK.

CREATING A NEW TABLE IN ACCESS

Suppose you are managing a project that involves both Excel and Access, and you need to add a new table to the Access database. You can do that with the following macro, and from there if need be, using the first macro in this lesson named AppendRecords, you can transfer any records you may have accumulated for that new table.

In this example, you create a simple three-field table to maintain a company's Employee Identification Number, which is a Primary Field, and the employees' last names and first names. The new table is named `tblEmployees`, and it is added to the `Database2.accdb` file that's been the subject of this lesson. Figure 32-5 shows `Database2.accdb` with the new table added after running the following macro named `CreateAccessTable`:

```
Sub CreateAccessTable()
    'Create a three-column table in an existing Access database:
    'EmployeeID
    'LastName
    'FirstName

    'Declare variables.
    Dim dbConnection As ADODB.Connection
    Dim dbCommand As ADODB.Command
    Dim dbFileName As String
    'Define the Access database path and name.
    dbFileName = "C:\Your\File\Path\Database2.accdb"
    'Set the assignment to open the connection.
    Set dbConnection = New ADODB.Connection

    'Define the Provider and open the connection.
    With dbConnection
        .Provider = "Microsoft.ACE.OLEDB.12.0;Data Source=" & dbFileName & _
        ";Persist Security Info=False;"
        .Open dbFileName
    End With

    'Set the Command variables.
    Set dbCommand = New ADODB.Command
    Set dbCommand.ActiveConnection = dbConnection

    'Create the table, which will be named tblEmployees.

```

```

dbCommand.CommandText = _
"CREATE TABLE tblEmployees (EmployeeID Char(10) " & _
"Primary Key, LastName text, FirstName text)"

'Execute the command to create the table.
dbCommand.Execute , , adCmdText

'Release Object variable memory.
Set dbCommand = Nothing
Set dbConnection = Nothing
End Sub

```

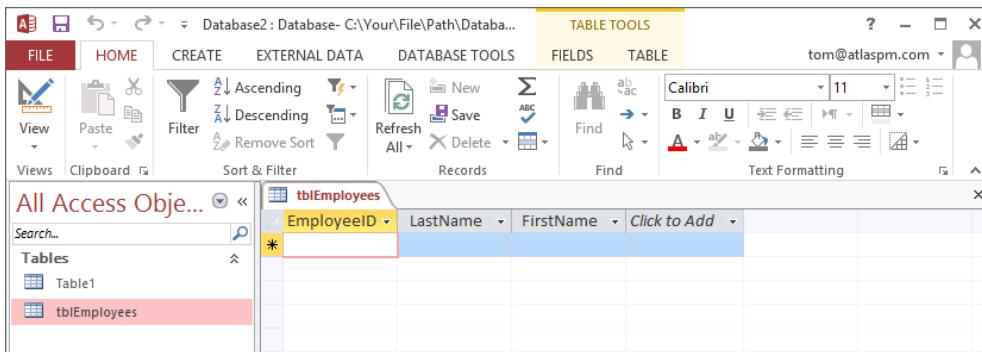


FIGURE 32-5

NOTE The text reference following the field names in the CommandText statement is to advise Access that the fields' data type will be Text. As you may know, with Access tables, other field types are Memo, Number, Date/Time, Currency, Yes/No, OLE Object, Hyperlink, and Attachment.

TRY IT

For this lesson, you maintain an Access database named `Database2.accdb`. In that database is a table named `tblEmployees`, for which you write a macro that adds a new field to hold the middle names of employees.

Lesson Requirements

To get the sample Excel workbook and Access database, you can download Lesson 32 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. In your Excel workbook, press Alt+F11 to go to the Visual Basic Editor.

2. From the VBE menu bar, click Insert \Rightarrow Module.
3. In the new module, type the name of your macro: **AddNewField**. Press Enter, and VBA automatically places a pair of parentheses after the macro name, followed by an empty line, followed by the `End Sub` statement. Your code looks as follows:

```
Sub AddNewField()
```

```
End Sub
```

4. Similar to what you have seen in this lesson's macros, declare three variables: one for the ADO connection, one for the ADO command, and one for the full path and name of the Access database you are working with:

```
Dim dbConnection As ADODB.Connection  
Dim dbCommand As ADODB.Command  
Dim dbFileName As String
```

5. Define the Access database path and name:

```
dbFileName = "C:\Your\File\Path\Database2.accdb"
```

6. Set the assignment to open the connection:

```
Set dbConnection = New ADODB.Connection
```

7. Define the Provider and open the connection:

```
With dbConnection  
.Provider = "Microsoft.ACE.OLEDB.12.0;Data Source=" & dbFileName & _  
";Persist Security Info=False;"  
.Open dbFileName  
End With
```

8. Set the Command variables:

```
Set dbCommand = New ADODB.Command  
Set dbCommand.ActiveConnection = dbConnection
```

9. Establish the command that adds a field for a middle name:

```
dbCommand.CommandText = _  
"ALTER TABLE tblEmployees Add Column MiddleName text)"
```

10. Execute the command to create the new field:

```
dbCommand.Execute , , adCmdText
```

11. Release Object variable memory:

```
Set dbCommand = Nothing  
Set dbConnection = Nothing
```

12. Examine the `Database2.accdb` file to confirm the existence of your new field for a middle name. Figure 32-6 shows what you should see, and the following code shows the complete macro.

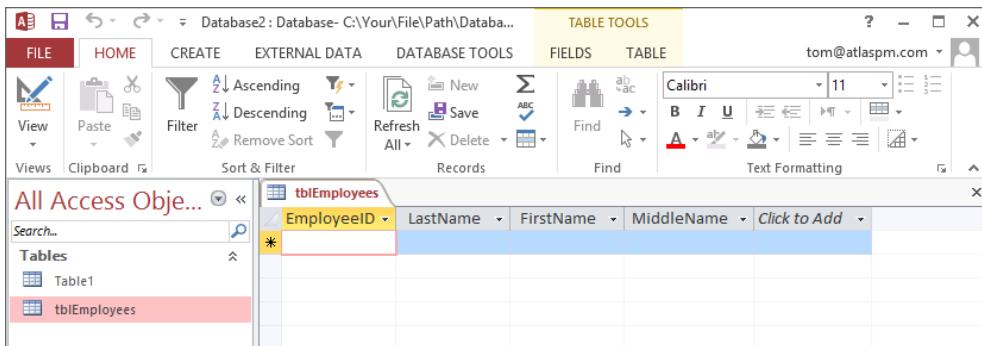


FIGURE 32-6

```

Sub AddNewField()

    'Declare variables
    Dim dbConnection As ADODB.Connection
    Dim dbCommand As ADODB.Command
    Dim dbFileName As String

    'Define the Access database path and name.
    dbFileName = "C:\Your\File\Path\Database2.accdb"

    'Set the assignment to open the connection.
    Set dbConnection = New ADODB.Connection

    'Define the Provider and open the connection.
    With dbConnection
        .Provider = "Microsoft.ACE.OLEDB.12.0;Data Source=" & dbFileName & _
        ";Persist Security Info=False;"
        .Open dbFileName
    End With

    'Set the Command variables.
    Set dbCommand = New ADODB.Command
    Set dbCommand.ActiveConnection = dbConnection

    'Establish the command that adds a field for a middle name.
    dbCommand.CommandText = _
    "ALTER TABLE tblEmployees Add Column MiddleName text)"

    'Execute the command to create the new field.
    dbCommand.Execute , , adCmdText

    'Release Object variable memory.
    Set dbCommand = Nothing
    Set dbConnection = Nothing

End Sub

```

REFERENCE Please select the video for Lesson 32 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

33

Working with PowerPoint from Excel

With each new release of its Office suite, Microsoft has made it increasingly easier to share information between applications. Copying data from Excel, such as a worksheet range or chart, and pasting it into a PowerPoint slide is as simple as copying and pasting from Excel into a Word document.

Still, PowerPoint is a unique animal in that its primary purpose is not to manipulate data but to display images of data for presentation purposes. When you need to transfer data from Excel to PowerPoint, such as a chart or a range of cells, I recommend you use VBA's `CopyPicture` method, which pastes an image of the data—not the data itself—into PowerPoint.

CREATING A NEW POWERPOINT PRESENTATION

Creating a new PowerPoint presentation file is an uncomplicated process; all you do is follow the usual steps for creating the PowerPoint application and then add a presentation with the expression `Presentations.Add`. Here's an example from start to finish, ending up with a new presentation file and an initial slide:

```
Sub CreateNewPresentation()

    'Declare Object variables for the PowerPoint application
    'and for the PowerPoint presentation file.
    Dim ppApp As Object, ppPres As Object
    'Declare Object variable for a PowerPoint slide.
    Dim ppSlide As Object

    'Open PowerPoint.
    Set ppApp = CreateObject("PowerPoint.Application")
    'Make the PowerPoint application visible.
    ppApp.Visible = msoTrue
```

```

'Create a new Presentation and add a slide.
Set ppPres = ppApp.Presentations.Add
With ppPres.Slides
    '11 is the numeric Constant for ppLayoutTitleOnly.
    'The Constant is used with late binding.
    Set ppSlide = .Add(.Count + 1, 11)
End With

'Save your new file.
ppPres.SaveAs Filename:=ThisWorkbook.Path & "\CreateTest.pptx"

'Release system memory reserved for the Object variables.
Set ppApp = Nothing
Set ppPres = Nothing
Set ppSlide = Nothing

End Sub

```

COPYING A WORKSHEET RANGE TO A POWERPOINT SLIDE

Now that you have just created a PowerPoint presentation file, while it's still open, suppose you want to copy a worksheet range into that presentation's first slide. The following macro uses an InputBox for the user to select a range to be copied.

NOTE Please note that this macro relies on the PowerPoint presentation file to be open. The code will not copy an Excel worksheet range to a closed PowerPoint presentation.

```

Sub CopyRange()
    'Declare a Range type variable.
    Dim rng As Range

    'Use an Application InputBox to have the user select the desired range.
    'Exit the macro if the user cancels.
    On Error Resume Next
    Set rng = Application.InputBox("Select a range to be copied:", Type:=8)
    If Err.Number <> 0 Then
        Err.Clear
        MsgBox "You did not enter a range.", vbInformation, "Cancelled"
        Exit Sub
    End If

    'Monitor the size of the range so an unreasonably large range is not attempted.
    If rng.Columns.Count > 6 Or rng.Rows.Count > 20 Then
        MsgBox "You selected a range that is too large." & vbCrLf & _
            "Please select a range that has no more than" & vbCrLf & _
            "6 columns and/or 20 rows.", vbCritical, "Range too large!"
        Exit Sub
    End If

```

```

'Declare Object variables.
Dim ppApp As Object, ppPres As Object, ppSlide As Object
'Assign the PowerPoint application you are working in to the ppApp variable.
Set ppApp = GetObject(, "Powerpoint.Application")
'Assign the presentation file you are working in.
Set ppPres = ppApp.ActivePresentation
Set ppSlide = ppPres.Slides(ppApp.ActiveWindow.Selection.SlideRange.SlideIndex)

'Copy the range as a picture.
rng.CopyPicture Appearance:=xlScreen, Format:=xlPicture
'Paste the picture of the range onto the slide.
ppSlide.Shapes.Paste.Select
'Align the range picture to be centered in the slide.
With ppApp.ActiveWindow.Selection.ShapeRange
.Align msoAlignCenters, msoTrue
.Align msoAlignMiddles, msoTrue
End With

'Release system memory reserved for the Object variables.
Set rng = Nothing
Set ppApp = Nothing
Set ppPres = Nothing
Set ppSlide = Nothing
End Sub

```

NOTE One consideration to monitor is the selected size of a range, as you can see in the code. My column and row limitations are just for example purposes. Whatever limit, if any, that you decide, the objective should be to place a clear, concise image on the slide.

COPYING CHART SHEETS TO POWERPOINT SLIDES

The Try It section of this lesson discusses how to copy an embedded chart into PowerPoint. If you have a choice between copying embedded charts or chart sheets, choose embedded charts—they provide you with greater control over how well they can be sized to fit a PowerPoint slide. This is because the ChartObject object is the container for an embedded chart, and it has properties that you can control for height, width, and location (where you can place it on the worksheet). Charts on chart sheets do not allow you to control their size.

Sometimes you won't have a choice, such as when a project calls for chart sheets to be copied into PowerPoint, and that is what the following macro accomplishes. To take things a step further, this macro does the following:

1. Creates a new PowerPoint presentation.
2. Adds an initial title slide.

3. Loops through all chart sheets, and with each one, copies its image and pastes it into a new slide.
4. Places a header title on each slide, then populates it with the chart name and formats the text.
5. Saves the file.

```
Sub CopyChartSheets()
'Declare Object variables for the PowerPoint application
'and for the PowerPoint presentation file.
Dim ppApp As Object, pptPres As Object
'Declare Object variable for a PowerPoint slide.
Dim pptSlide As Object
'Declare variables for the Charts you will copy.
Dim ch As Chart
'Declare an Integer type variable for a running count of slides
'as each chart sheet is added to the new presentation file.
Dim SlideCount As Integer

'Open PowerPoint.
Set ppApp = CreateObject("PowerPoint.Application")
'Make the PowerPoint application visible.
ppApp.Visible = msoTrue

'Create a new Presentation and add a title slide.
Set pptPres = ppApp.Presentations.Add
With pptPres.Slides
Set pptSlide = .Add(.Count + 1, 11)
End With
pptSlide.Shapes.Title.TextFrame.TextRange.Text = "Chart sheet copy test"

'Open a For...Next loop to place each Chart sheet in a slide.
For Each ch In ThisWorkbook.Charts
ch.CopyPicture Appearance:=xlScreen, Format:=xlPicture, Size:=xlScreen

'Add a new slide.
SlideCount = pptPres.Slides.Count
Set pptSlide = pptPres.Slides.Add(SlideCount + 1, 11)
ppApp.ActiveWindow.View.GotoSlide pptSlide.SlideIndex
'Paste and select the chart picture.
pptSlide.Shapes.Paste

>Select the pasted shape.
pptSlide.Shapes(1).Select

'Align the chart to be centered in the slide.
With ppApp.ActiveWindow.Selection.ShapeRange
.Align msoAlignCenters, msoTrue
.Align msoAlignMiddles, msoTrue
End With

'Set the position of the slide's header label.
With ppApp.ActiveWindow.Selection
.SlideRange.Shapes.AddLabel _
```

```

(msoTextOrientationHorizontal, 300, 20, 500, 50).Select
.ShapeRange.TextFrame.WordWrap = msoFalse
'Format the header label.
With .ShapeRange.TextFrame.TextRange
.Characters(Start:=1, Length:=0).Select
.Text = "This is " & ch.Name
With .Font
.Name = "Arial"
.Size = 12
.Bold = msoTrue
End With
End With
End With

'Continue the loop until all chart sheets have been copied.
Next ch

'End the macro by activating the first slide.
ppApp.ActiveWindow.View.GotoSlide 1

'Save your new file.
pptPres.SaveAs Filename:=ThisWorkbook.Path & "\ChartSheetTest.pptx"

'Release system memory reserved for the Object variables.
Set ppApp = Nothing
Set pptSlide = Nothing
Set pptPres = Nothing
Set ppApp = Nothing

End Sub

```

RUNNING A POWERPOINT PRESENTATION FROM EXCEL

Running a PowerPoint presentation from Excel provides a dynamic effect to your Excel project. Unlike Word, Outlook, or Access, just opening a presentation file in PowerPoint is not enough if you want to show what that file contains. You can cycle through the slides with the `slideShowSettings.Run` statement. Notice the `With` structure that demonstrates a method of setting the amount of time (three seconds of the `advancetime` property in this example) that each slide is shown, without affecting the user's local PowerPoint slide transition settings:

```

Sub PowerPointSlideshow()

'Declare Object variables for the PowerPoint application
'and for the PowerPoint presentation file.
Dim ppApp As Object, pptPres As Object
'Declare String variables for folder path and name of file.
Dim strFilePath As String, strFileName As String
'Define the String variables with the directory path and name.
strFilePath = "C:\Your\File\Path\
strFileName = "PowerPointExample1.pptx"

'Verify if the path and filename really exist.
'If not, exit the macro and advise the user.

```

```
If Dir(strFilePath & strFileName) = "" Then
    MsgBox _
        "The PowerPoint file " & strFileName & vbCrLf & _
        "does not exist in the folder path" & vbCrLf & _
        strFilePath & ".", _
        vbInformation, "No such animal."
    Exit Sub
End If

'Open PowerPoint.
Set ppApp = CreateObject("PowerPoint.Application")
'Make the PowerPoint application visible.
ppApp.Visible = msoTrue
'Open the PowerPoint presentation you want to run.
Set ppPres = ppApp.Presentations.Open(strFilePath & strFileName)

'Establish the amount of time each slide should be shown
'which in this example is 3 seconds.
With ppPres.Slides.Range.SlideShowTransition
    .AdvanceOnTime = True
    .AdvanceTime = 3
End With

'Run the PowerPoint presentation.
ppPres.SlideShowSettings.Run
'When the presentation is completed, have VBA regard it as saved
'so you are not prompted to save the presentation when you close it.
ppPres.Saved = True
'Quit Powerpoint (optional)
'ppApp.Quit

'Release memory taken from the Object variables.
Set ppPres = Nothing
Set ppApp = Nothing

End Sub
```

TRY IT

In this lesson, you copy an embedded chart from a worksheet and paste its picture image into an empty slide in an open PowerPoint presentation.

Lesson Requirements

To get the sample workbook you can download Lesson 33 from the book's website at www.wrox.com/go/excelvba24hour.

Step-by-Step

1. From your workbook, press Alt+F11 to go to the Visual Basic Editor.
2. From the menu bar at the top of the VBE, click Insert ➔ Module.

3. In the module you just created, type `Sub CopyEmbeddedChart` and press Enter. VBA automatically places a pair of empty parentheses at the end of the `Sub` line, followed by an empty line, and the `End Sub` line below that. Your macro looks like this so far:

```
Sub CopyEmbeddedChart()  
End Sub
```

4. This example assumes you have PowerPoint open, with your destination presentation file open, and your destination slide selected. Declare variables for the PowerPoint application, presentation filename, and `Slide` object:

```
Dim ppApp As Object, ppPres As Object, ppSlide As Object
```

5. For this example, you want to copy the first chart on your worksheet. Select the chart programmatically by its index number 1:

```
ActiveSheet.ChartObjects(1).Select
```

6. Establish the identity of the open PowerPoint application:

```
'Establish the identity of the open PowerPoint application.  
'Check to make sure PowerPoint is open.  
'If PowerPoint is not open, halt the macro and inform the user.  
On Error Resume Next  
Set ppApp = GetObject(, "Powerpoint.Application")  
If Err.Number = 429 Then  
    Err.Clear  
    MsgBox "Please open PowerPoint first," & vbCrLf & _  
        "and open the presentation where" & vbCrLf & _  
        "you want to paste the copied chart.", 48, "Cannot continue."  
    Range("A1").Select  
    Exit Sub  
End If
```

7. Establish the identity of the open PowerPoint presentation:

```
Set ppPres = ppApp.ActivePresentation
```

8. Establish a reference to the destination slide you have manually selected:

```
Set ppSlide = ppPres.Slides(ppApp.ActiveWindow.Selection.SlideRange.SlideIndex)
```

9. Copy the selected chart:

```
ActiveChart.CopyPicture Appearance:=xlScreen, Size:=xlScreen, Format:=xlPicture
```

10. Paste the chart into the PowerPoint slide:

```
ppSlide.Shapes.Paste
```

11. Select the picture of the chart you just pasted onto the PowerPoint slide:

```
ppSlide.Shapes(1).Select
```

12. Align the chart picture to be centered in the slide:

```
With ppApp.ActiveWindow.Selection.ShapeRange  
.Align msoAlignCenters, msoTrue  
.Align msoAlignMiddles, msoTrue  
End With
```

13. Deselect the selected chart:

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

14. Release system memory reserved for the Object variables:

```
Set ppApp = Nothing  
Set ppPres = Nothing  
Set ppSlide = Nothing
```

15. When completed, the macro looks like this, with comments that have been added to explain each step:

```
Sub CopyEmbeddedChart()  
'This example assumes you have PowerPoint open,  
'with your destination presentation file open,  
'and your destination slide selected.  
  
'Declare variables for the PowerPoint application,  
'presentation filename, and Slide object.  
Dim ppApp As Object, ppPres As Object, ppSlide As Object  
  
'For this example, you want to copy the first chart on your worksheet.  
'Select the chart by its index number one.  
ActiveSheet.ChartObjects(1).Select  
  
'Establish the identity of the open PowerPoint application.  
'Check to make sure PowerPoint is open.  
'If PowerPoint is not open, halt the macro and inform the user.  
On Error Resume Next  
Set ppApp = GetObject(, "Powerpoint.Application")  
If Err.Number = 429 Then  
Err.Clear  
MsgBox "Please open PowerPoint first," & vbCrLf &  
"and open the presentation where" & vbCrLf &  
"you want to paste the copied chart.", 48, "Cannot continue."  
Range("A1").Select  
Exit Sub  
End If  
  
'Establish the identity of the open PowerPoint presentation.  
Set ppPres = ppApp.ActivePresentation  
'Establish a reference to the destination slide you have manually selected.  
Set ppSlide = ppPres.Slides(ppApp.ActiveWindow.Selection.SlideRange.SlideIndex)  
  
'Copy the selected chart.  
ActiveChart.CopyPicture Appearance:=xlScreen, Size:=xlScreen, Format:=xlPicture
```

```
'Paste the chart into the PowerPoint slide.  
ppSlide.Shapes.Paste  
'Select the picture of the chart you just pasted onto the PowerPoint slide.  
ppSlide.Shapes(1).Select  
  
'Align the range picture to be centered in the slide.  
With ppApp.ActiveWindow.Selection.ShapeRange  
.Align msoAlignCenters, msoTrue  
.Align msoAlignMiddles, msoTrue  
End With  
  
'Deselect the selected chart.  
Range("A1").Select  
  
'Release system memory reserved for the Object variables.  
Set ppApp = Nothing  
Set ppPres = Nothing  
Set ppSlide = Nothing  
  
End Sub
```

16. Press Alt+Q to return to the worksheet. Press Alt+F8 to show the Macro dialog box, and test the macro by selecting the macro name and clicking the Run button.

REFERENCE Please select the video for Lesson 33 online at www.wrox.com/go/excelvba24hour. You will also be able to download the code and resources for this lesson from the website.

INDEX

A

A1 references in formulas, 113–115
vs. R1C1, 115
toggling with R1C1, 116–118
absolute references
 converting to relative, 121–122
 named ranges with, 119–120
Access application and tables
 adding records to, 419–423
 creating, 426–427
 exporting, 423–426
 QueryTables for, 356–358
 step-by-step example, 427–429
accessing
 UserForms, 281
 VBA environment, 11–15
AccessToExcel macro, 423–426
activate events
 workbooks, 154, 157
 worksheet, 143
Activate method, 37, 52
ActivateWord macro, 400–401
activating
 objects, 37
 Word documents, 399–401
 workbooks, 67–68
 worksheets, 52
active elements, coloring,
 373–375

active worksheet and workbook names,
 243–244
ActiveCell object, 69
ActiveConnection property
 Command object, 368
 Recordset object, 367
ActiveX controls
 collections of, 328
 CommandButtons, 187–191
 Control Toolbox, 186–187
 vs. Form controls, 182
 overview, 181
ActiveX Data Objects (ADO)
 Command object, 368
 Connection object, 367
 introduction, 365–367
 Recordset object, 367–368
 with SQL, 368–372
add-ins
 benefits, 336
 closing, 349
 code changes, 348
 converting files to, 341–342
 creating, 336–340
 description, 335
 installing, 342–346
 removing, 349–350
 step-by-step example, 350–352
 user interface, 346–348
Add-Ins dialog box, 343–346, 349, 351

Add method
charts, 201
PowerPoint presentations, 431
Word documents, 402
workbooks, 67
worksheets, 68–69
Add Watch dialog box, 262
AddCorrection macro, 108
AddFiveWorksheets macro, 102–103
AddItem method
 ComboBox controls, 292
 ListBox controls, 291
AddNewField macro, 428–429
addresses, extracting from hyperlinks, 242
AddSheetTest macro, 264–265
AddWorkbooks macro, 68
ADO. See ActiveX Data Objects (ADO)
AdvancedFilter
 deleting rows with duplicates, 161–164
 unique lists from columns, 167–168
Alignments button, 306
Alt+= keys, 193
Alt+D keys, 426
Alt+F1 keys, 204
Alt+F8 keys, 22, 39, 246
Alt+F11 keys, 25–26
Alt+O+E keys, 204
Alt+Q keys, 30
Alt+T+I keys, 344
American Standard Code for Information
 Interchange (ASCII), 289
Analysis ToolPak add-in, 346
Analysis ToolPak VBA add-in, 346
AND logical operators, 86
apostrophes (') for comments, 36, 38
AppendRecords macro, 421–423
Application.Caller statement
 Form controls, 184–186
 UDFs, 240
Application object, 50, 67

Application.Volatile statement, 243–244
applications, variable scope in, 63
arguments in UDFs, 239
arrays
 benefits, 128–129
 boundaries, 132
 declaring, 129–130
 dynamic, 133–134
 with fixed elements, 132–133
formulas, 120–122
Option Base statement, 130–131
overview, 127–128
step-by-step example, 134–136
ArraySheets macro, 132
ArrayTest macro, 131
ArrayWeekdays macro, 132–133
As keyword, 55
ASCII (American Standard Code for
 Information Interchange), 289
Assign Macro dialog box, 183–184, 197–198
assigning
 shortcut keys, 19, 36
 values to variables, 56–57
asterisks (*) in SELECT statement, 369
Auto List Members option, 71–72
AutoCorrect list, updating, 108
automatically run macros, 5–6
automation, Office. See Office automation
AverageBowlingScores macro, 120

B

BackColor property, 327
BASIC (Beginner's All-purpose Symbolic
 Instruction Code) programming
 language, 4
binding in Office automation, 392–394
Boolean data type, 58
boundaries for arrays, 132
Break button, 255

breakpoints, 259–261
 Bring to Front button, 306
 bugs. See debugging code
 BuildDynamicString macro, 412
 buttons
 Form controls, 183–184
 message boxes, 93
 bypassing errors, 265–266
 Byte data type, 58

C

calculate events, 144
 CalculateSalary macro, 120
 Call Stack dialog box, 263
 calling UDFs from macros,
 245–246
 Caption property
 CommandButtons, 188, 286
 Label controls, 276, 287
 UserForms, 274
 Case keyword, 91–92
 Cell object, 50, 51
 cells
 clearing, 70
 color, 23–24
 coloring, 373–376
 data validation, 383–387
 filling, 118–119
 logging changes to, 380
 ranges. See ranges and Range object
 summing numbers in, 239–240
 Cells property, 76
 Centering button, 306
 centuries, entering, 59
 Change Chart Type dialog box, 200
 change events
 workbooks, 154–155
 worksheets, 141–142, 144–148
 CHAR function, 289

characters, extracting from strings, 241–242
 Chart object, 199
 chart sheets
 adding charts to, 200–202
 copying to slides, 433–435
 ChartLocation macro, 84
 ChartObject object, 199
 charts, 199
 adding to chart sheets, 200–202
 adding to worksheets, 202–204
 deleting, 207–208
 locating, 82–84, 209
 looping through, 206–207
 moving, 204–205
 PivotCharts, 223–226
 renaming, 208
 step-by-step example, 208–211
 UserForms, 314–315
 Charts collection, 53, 199
 ChartSheetsToWorkbook macro, 205
 ChartSheetToWorksheet macro, 204–205
 CheckBox controls
 color, 329–330
 overview, 294–295
 CheckBox1_Click macro, 295
 class modules, 28, 321
 benefits, 323–326
 classes, 321–322
 collections, 326
 description, 322–323
 objects
 creating, 323
 embedded, 326–330
 step-by-step example, 330–334
 ClearClipboard macro, 381–382
 ClearContents method, 51–52, 70
 ClearData macro, 183, 185
 clearing
 clipboard, 381–382
 ranges, 51–52, 70

click events
 CommandButtons, 286
 workbooks, 155–156
 worksheets, 142
clipboard, clearing, 381–382
Close button, disabling, 307–308
close events, 154
Close method
 Connection object, 367
 Recordset object, 368
CloseAllOtherWorkbooks macro, 68
CloseOneWorkbook macro, 105
CloseOneWorkbookFaster macro, 105
CloseWorkbooks macro, 104
closing
 add-ins, 349
 connections, 367
 Recordset objects, 368
 UserForms, 281–283
 workbooks, 68, 104–105
 worksheets, 104
cmdButtonGroup_Click macro, 327
cmdButtonGroup_MouseMove macro, 327
cmdCancel_Click macro, 286
cmdContinue_Click macro, 308
cmdLandscape_Click macro, 286
cmdOK_Click macro
 add-in example, 338–339
 checkboxes, 296
cmdPortrait_Click macro, 286
cmdSortDown_Click macro, 311–312
cmdSortUp_Click macro, 311
code
 debugging. See debugging code
 macros, 36
 UserForms, 281
Code window, 27
Collection object, 52–53
collections
 ActiveX controls, 328
 creating, 326
 For...Each...Next loops, 104
 object model, 52–53
 step-by-step example, 71–73
 workbooks, 67–69
colon character (:) in Select Case structure, 92
color
 cells, 23–24
 CheckBox controls, 329–330
 comments, 36, 38
 hex codes, 327
Color property, 51
colored cells, summing numbers in, 239–240
coloring
 active elements, 373–375
 cells, 375–376
columns
 coloring, 373–375
 last, 80–81
ComboBox controls
 overview, 292–294
 populating, 312–314
 pre-sorting items in, 310–311
ComboBox1_Change macro, 313
Command object, 368
CommandBar object, 346
CommandButton controls
 ActiveX controls, 187–191
 adding, 278–280
 overview, 286
CommandButton1_Click macro
 ActiveX controls, 190
 hiding columns, 288
 OptionButtons, 297–298
 summing numbers, 290
CommandButton4_Click macro, 300
CommandText property
 Access fields, 427
 Command object, 368
CommandType property, 368

commas (,)
arguments, 239
ranges, 77
thousands separators, 16
variable declarations, 59

Comment2Text macro, 360

comments
cell change logs, 380
conditional formatting for, 244–245
listing unique items, 169–170
in macros, 36–39

Comments collection, 52

compatibility of macros, 34

conditional formatting in UDFs, 244–245

ConfirmExample macro, 93

Connection object, 367

ConnectionString property, 367

constants, 63–64

continuously populated ranges, 75–77

Control Toolbox, 186–187

controls, 274–280
Application.Caller for, 184–186
Buttons, 183–184
CheckBox, 294–295
ComboBox, 292–294
CommandButton, 187–191, 286
Control Toolbox, 186–187
Form and ActiveX, 181–182
Forms toolbar, 182–183
Frame, 298–300
frequently used, 285
Label, 287–288
ListBox, 290–292
MultiPage, 300–301
OptionButton, 296–298
step-by-step examples, 191–198,
301–304
TextBox, 288–290

ControlSource property, 288

ConvertAbsoluteToRelative macro, 121–122

converting
absolute and relative references, 121–122
files to add-ins, 341–342

ConvertRelativeToAbsolute macro, 121

CopyChartSheets macro, 434–435

CopyEmbeddedChart macro, 437–439

copying
chart sheets to slides, 433–435
to clipboard, 381–382
ranges
to PowerPoint presentations,
432–433
to Word documents, 402–403

CopyRange macro, 432–433

CountFormulas macro, 122

Create PivotTable dialog box, 213

CreateAccessTable macro, 426–427

CreateChartSameSheet macro, 202–203

CreateChartSheet macro, 200

CreateNewPresentation macro, 431–432

CreatePivot macro, 227

CreatePivotChart macro, 234–235

CreateTextFiles macro, 359

CreateWordDoc macro, 402

Ctrl+Alt+F9 keys, 243

Ctrl+Break keys, 243

Ctrl+F11 keys, 26

Ctrl+G keys, 28, 72

Ctrl+R keys, 26, 150

Ctrl+S keys, 350

Ctrl+Shift+Enter keys, 120

Ctrl+Shift+F9 keys, 261

Currency data type, 58

current cells, coloring, 375–376

CurrentQuarter macro, 92

CurrentRegion property
charts, 200–201
overview, 76–77

Custom Lists dialog box, 384–385

Customize Ribbon option, 15

CustomListDV macro, 385–387

CutCopyMode property, 381

D

DAO (Data Access Objects) library, 366

data access, ADO. See ActiveX Data Objects (ADO)

data ranges, identifying, 79

data types, 55

arrays, 127

dates and time, 58–59

declaring, 59–61

overview, 57–58

data validation in cells, 383–387

database management system (DBMSs), 366

databases

Access. See Access application and tables terms, 366

DataRangeLastRowsColumns macro, 80–81

dates and Date data type

declaring, 58–59

description, 58

filtering, 376–379

querying, 361–364

DateSerial function, 111, 376

DBMSs (database management system), 366

deactivate events

workbooks, 154, 157–158

worksheet, 144

Debug toolbar, 254–255

Break button, 255

Design Mode button, 255

Reset button, 255

Run button, 255

stepping through code, 255–256

Step Into button, 257–258

Step Out button, 259

Step Over button, 258–259

Toggle Breakpoint button, 259–261

debugging code

Call Stack dialog box, 263

Debug toolbar. See Debug toolbar

errors

bypassing, 265–266

causes, 252–254

handling, 264–265

Immediate window, 261–262

Locals window, 261

overview, 251–252

Quick Watch window, 263

step-by-step example, 266–268

Watch window, 262–263

Decimal data type, 58

decisions, 85

If...Then statements, 88

If...Then...Else statements, 89

If...Then...ElseIf statements, 90

IIF statements, 90–91

logical operators, 85–88

Select Case structure, 91–92

step-by-step example, 94–97

user, 92–94

declaring

arrays, 129–130

dynamic, 133–134

with fixed elements, 132–133

constants, 63–64

variables, 55–56, 59–61

DELETE statement in SQL, 370–371

DeleteAllPivotTablest macro, 232

DeleteAndCreate macro, 361

DeleteArrayColors macro, 167

DeleteChartSheets macro, 208

DeleteDuplicatesColumnA macro, 162

DeleteDuplicatesColumnD macro, 162–163

DeleteDuplicateRecords macro, 164–165

DeleteRows3YearsOld macro, 378–379

- deleting
- charts, 207–208
 - hyperlinks, 261
 - macros, 39
 - modules, 42–43
 - PivotTables, 232
 - rows
 - with duplicates, 161–167
 - filtered dates, 378–379
 - in SQL, 370–371
 - descriptions
 - in Insert Function dialog box, 246–248
 - for macros, 19
 - Design Mode button, 255
 - Design mode in Control Toolbox, 188, 191
 - Developer tab, 13–15
 - Dim statement, 129–130
 - dimensions, arrays, 129
 - Dir function, 107
 - disabling
 - Close button, 307–308
 - Frames, 298–299
 - worksheet events, 139–140
 - DisplayGridlines property, 88
 - displaying
 - photographs, 308–309
 - real-time charts, 314–315
 - Do...Loop Until loops, 109
 - Do...Loop While loops, 109
 - Do Until loops, 107–108
 - Do While loops, 106–107
 - double-click events
 - workbooks, 155–156
 - worksheets, 142
 - Double data type, 58
 - DoWhileExample macro, 107
 - duplicates
 - deleting rows with, 161–167
 - selecting range of, 171–172
 - step-by-step example, 173–179
 - unique lists from multiple columns, 167–170
 - dynamic arrays, 133–134
 - dynamic last rows and columns, 80–81

E

- e-mail
- creating, 410–411
 - example, 413–414
 - step-by-step example, 415–418
 - transferring ranges to, 411–413
 - worksheets, 415
 - early binding, 392–395
 - EarlyBindingTest macro, 393
 - editing macros, 37–39
 - efficiency, variables for, 57
 - elements, array, 127
 - EmailAttachmentRecipients macro, 416–418
 - EmailSingleSheet macro, 415
 - embedded charts
 - adding to worksheets, 202–204
 - copying to PowerPoint, 436–439
 - looping through, 206–207
 - moving, 204–205
 - embedded form controls. See Form controls
 - embedded objects, class modules for, 326–330
 - EmbeddedChartToAnotherWorksheet macro, 205
 - EmbeddedChartToChartSheet macro, 205
 - EmptyRecycleBin function, 382
 - EnableEvents property, 139–140
 - enabling worksheet events, 139–140
 - End Function statements, 238
 - End If statements, 88
 - end of ranges, 81–82

errors

debugging. See debugging code
UDFs, 242
Word applications, 400
Euro Currency Tools add-in, 346
events, 137
 automatically run macros, 5–6
 CommandButton, 187–191
 Object Browser, 29
 workbook. See workbook events
 worksheet. See worksheet events
ExampleEmail macro, 413–414
Excel Options dialog box
 add-ins, 344
 for Developer tab, 13–14
 formulas, 116
 lists, 384–385
 Option Explicit statement, 61
Exit For statements, 105
exiting
 For loops, 104–105
 VBE, 30
ExportFromExcelToWord macro, 403
exporting Access tables, 423–426
expressions in Watch window, 262–263
Extended ASCII characters, 289
external data, 353
 ADO. See ActiveX Data Objects (ADO)
 QueryTables
 for Access, 356–358
 from web queries, 353–356
step-by-step example, 361–364
text files for, 359–361
extracting
 addresses from hyperlinks, 242
 characters from strings, 241–242
ExtractLetters UDF, 241–242
ExtractNumbers UDF, 241, 246–247

F

F2 key, 23, 243
F4 key, 337
F5 key, 23, 70
F9 key, 260
F11 key, 202
False value in truth tables, 85–88
FavoriteMovies macro, 127
FavoriteMoviesLoop macro, 128
FavoriteMoviesRange macro, 128–129
field lists in PivotTables, hiding, 217–219
fields, database, 366
files, converting to add-ins, 341–342
FillBlankCellsFromAbove macro,
 118–119
FilterBetweenDates macro, 376–378
FilterDateAfterToday macro, 378
FilterDateBeforeToday macro, 378
filters
 AdvancedFilter, 161–164, 167–168
 dates, 376–379
 deleting rows with duplicates, 161–164
 PivotTables, 214
Find_LastRow_LastColumn macro, 75
Find method
 error bypass structure for, 266–268
 ranges, 79
FindFormulas macro, 71
FindHello macro, 109
FindTest macro, 268
fixed elements, declaring arrays with,
 132–133
fixed-iteration loops, 102
For...Next loops, 102–103
For...Each...Next loops, 104
forcing variable declarations, 59–61
ForeColor property, 327

Form controls
 vs. ActiveX, 182
 Application.Caller, 184–186
 buttons, 183–184
 Control Toolbox, 186–187
 Forms toolbar, 182–183
 overview, 181
 step-by-step example, 191–198

Format Cells dialog box
 color, 23
 numbers, 193–194
 PivotTables, 220–221

formatting
 PivotTable numbers, 219–222
 UDFs, 244–245

forms. See UserForms

Forms toolbar, 182–183

FormulaArray method, 120

FormulaR1C1 method, 114, 116

formulas, 113
 array, 120–122
 counting, 122
 entering, 114–115
 references, 113–115
 A1 vs. R1C1, 115
 converting absolute and relative, 121–122
 mixed, in filling empty cells, 118–119
 named ranges, 119–120
 toggling between style views, 116–118
 step-by-step example, 124–126
 summing lists, 122–124

ForNextExample2 macro, 103

ForNextExample3 macro, 103

Frame controls, 276–277, 298–300

FROM clause in SELECT statements, 369

Function statement, 238
 functions. See user-defined functions (UDFs)

G

GetComment UDF, 249, 350
 GetObject function, 399–400
 GetTextMessage macro, 361
 Go To dialog box
 accessing, 23
 SpecialCells, 70, 244
 Go To Special dialog box, 23
 GroupName property, 298
 Groups button, 306
 grpCBX_Click macro, 329

H

Height parameter for charts, 209
 hex codes for color, 327
 Hide method for UserForms, 281
 hiding
 PivotTable field lists, 217–219
 UserForms, 283
 history of VBA, 4
 hyperlinks
 deleting, 261
 events, 142–143
 extracting addresses from, 242

I

icons, displaying, 12–13
 identifying ranges, 79–80
 If...Then statements, 88
 If...Then...Else statements, 89
 If...Then...ElseIf statements, 90
 IIF statements, 90–91
 Image controls, 309

Immediate window, 28, 31, 261–262
Import Data dialog box, 426
Importance property for e-mail, 411
ImportHistory macro, 356
importing
 Access tables, 423–426
 Word documents, 404–405
ImportStocks macro, 354–355
ImportToExcelFromWord macro, 404–405
indefinite loops, 102
index numbers
 arrays, 127–128, 131–132
 charts, 205
 lists, 383
 worksheets, 68–69, 107
Index property
 charts, 82
 PivotTables, 229
infinite loops, 140
Initialize events
 labels, 287
 ListBox controls, 291
input boxes, 94
InputPassword macro, 110
Insert Chart dialog box, 225
Insert Function dialog box, descriptions in, 246–248
inserting
 modules, 39–40
 rows
 on data changes, 172–173
 databases, 369–370
 input boxes for, 94
 loops for, 106
InsertRows macro, 94, 106
installing add-ins, 342–346
instantiating
 classes, 323
 objects, 325
Integer data type
 description, 58

variables, 55
IntelliSense tool, 71–73
interface for add-ins, 346–348
IsNumeric function, 146
iterations in loops, 101–102

J

JKP Application Development Services, 381

K

KeepOnlyArrayColors macro, 166–167

L

Label controls
 OptionButtons, 330–331
 overview, 287–288
 UserForms, 276–277
last rows and columns, finding, 80–81
late binding
 description, 394
 vs. early, 394–395
 step-by-step example, 395–397
LateBindingTest macro, 394
LBound function, 132
Left parameter for charts, 209
LEN function, 293
letters, extracting from strings, 241–242
liabilities of VBA, 8
libraries in Object Browser, 28–30
lifetime
 constants, 64
 variables, 61–63
Link UDF, 242
ListBox controls
 overview, 290–292
 populating, 312–314
 pre-sorting items in, 310–311
ListBox1_Click macro, 292

lists

- arrays as, 128–129
- custom, 385–387
- from multiple columns, 167–170
- summing, 122–124
- ListStyle** property, 290
- LoadPicture** dialog box, 309
- local macro scope, 62
- Locals** window, 261
- Location** property, 200
- Locked** property, 51
- locking VBE, 43–44
- logging cell changes, 380
- logical errors, 253–254
- logical operators, 85
 - AND, 86
 - NOT, 87–88
 - OR, 86–87
- Long data type, 58
- look and feel, simplifying, 7
- LoopAllChartSheets** macro, 207
- LoopAllEmbeddedCharts** macro, 206
- loops**
 - description, 101–102
 - Do...Loop Until**, 109
 - Do...Loop While**, 109
 - Do Until**, 107–108
 - Do While**, 106–107
 - embedded charts, 206–207
 - exiting, 104–105
 - For...Each...Next**, 104
 - For...Next**, 102–103
 - infinite, 140
 - nesting, 110–111
 - reverse, 105–106
 - step-by-step example, 111–112
 - types, 102
 - While...Wend**, 110
- LoopTwelveMonths** macro, 112

M

- Macro** dialog box, 21–22
- Macro Options** dialog box, 247–248
- Macro Recorder** limitations, 37
- macros**
 - automatically running, 5–6
 - buttons, 183–184
 - calling UDFs from, 245–246
 - code, 36
 - compatibility, 34
 - deleting, 39
 - description, 3–4
 - editing, 37–39
 - Form controls, 181
 - locating, 33–35
 - Macro Recorder**, 16–21
 - names, 19, 22
 - running, 21–22
 - step-by-step example, 22–24
 - UDFs called by, 238
 - variable scope in, 62
 - VBA environment access, 11–15
- MailItem** objects, 410–411
- maximizing UserForms, 308
- Me** keyword, 306
- message boxes, 92–93
- methods**
 - IntelliSense for, 71–73
 - Object Browser**, 29
 - object model, 49, 51–52
- MID** function, 243–244
- mixed references**
 - filling empty cells, 118–119
 - named ranges with, 119–120
- modal UserForms, 306–307
- modeless UserForms, 306–307

modules

- class. See class modules
- deleting, 42–43
- inserting, 39–40
- renaming, 41–42
- types, 28
- UDFs, 238
- UserForms, 281
- variable scope in, 62–63
- VBE, 34–35
- moving charts, 204–205
- MultiPage controls, 300–301
- multiple columns
 - deleting rows with duplicates, 164–167
 - unique lists from, 167–170
- MultiSelect property, 290, 292

N

- Name property
 - UserForms, 274
 - worksheets, 51
- named ranges, 119–120
- names
 - active worksheets and workbooks, 243–244
 - charts, 205, 208
 - macros, 19, 22
 - modules, 41–42
 - PivotFields, 230
 - testing, 265–266
 - UDFs, 239
 - variables, 55–56
- Names collection, 53, 70
- NameWB UDF, 244
- Naval Observatory, querying, 361–364
- nesting loops, 110–111
- New Formatting Rule dialog box, 245
- new sheet events, 156–157
- Next statements, 103
- non-Excel applications, controlling, 7–8
- noncontinuously populated ranges, 77

NOT logical operators, 87–88

number signs (#) for dates and time, 58–59

numbers

- extracting from strings, 241–242
- formatting in PivotTables, 219–222
- summing, 239–240

O

Object Browser, 28–30

Object data type, 58

object-oriented programming

introduction, 49

object model

- collections, 52–53
- methods, 51–52
- overview, 50–51
- properties, 51

summary, 53

objects

creating, 323

embedded, 326–330

IntelliSense for, 71–73

ODBC (Open Database Connectivity), 366

Office automation, 391

Access. See Access application and tables

benefits, 391–392

binding, 392–395

Outlook. See Outlook application and e-mail

PowerPoint. See PowerPoint presentations

step-by-step example, 395–397

Word. See Word application and documents

OFFSET property, 78

OLEObject keyword, 328

OLEObjects keyword, 328

On Error GoTo statements, 264

On Error Resume Next statements, 265, 400

OnKey procedures, 202

Open Database Connectivity (ODBC), 366

- open events, 153–154
Open method
 Connection object, 367
 Recordset object, 368
OpenAllFiles macro, 107
opening
 databases, 367, 424
 Outlook, 409–410
 PowerPoint, 395–397
 Recordset objects, 368
 Word documents, 400–401, 406–408
 workbooks, 107
OpenOrClosed UDF, 245–246
OpenOutlook macro, 409–410
OpenPowerPoint macro, 395–397
OpenRequestedWordDoc macro, 406–408
OpenTest UDF, 245
OptGroup_Click macro, 332
Option Base statement, 130–131
Option Explicit statement, 59–61
OptionButton controls
 adding, 277–278
 overview, 296–298
 step-by-step example, 330–334
Options dialog box
 Auto List Members, 71–72
 view style, 116–117
OR logical operators, 86–87
ORDER BY statement, 369
Outlook application and e-mail, 409
 creating, 410–411
 example, 413–414
 opening, 409–410
 step-by-step example, 415–418
 transferring ranges to, 411–413
 worksheets, 415
- P**
- page breaks, 379
PageBreakInsert macro, 379
- Parent property, 52
parentheses ()
 arrays, 129–130
 message boxes, 93
 Sub statement, 36
 UDFs, 239
PasswordChar property, 288
passwords
 entering, 110
 step-by-step example, 94–97
 UserForms, 288
 VBE, 43
PasswordTest macro, 97
photographs, 308–309
PickSixLottery macro,
 110–111
Picture property, 309
pie charts, 210–211
Pieterse, Jan Karel, 381
PivotCaches, 226–230
PivotCharts, 223–226
PivotFields, 230
PivotItems, 231
PivotTables, 52, 213
 creating, 213–217
 field list hiding, 217–219
 formatting numbers in,
 219–222
 PivotCaches, 226–230
 PivotCharts, 223–226
 PivotFields, 230
 pivoting data in, 222
 PivotItems, 231
 refreshing, 226, 232
 step-by-step example, 232–235
 workbook events, 156
 worksheet events, 144
PivotTables collections, 231–232
points, 209
populating ListBox and ComboBox items,
 312–314

- PowerPoint presentations
binding, 395–397
copying chart sheets to, 433–435
copying ranges to, 432–433
creating, 431–432
running, 435–436
- PowerPointSlideshow macro, 435–436
- pre-sorting ListBox and ComboBox items, 310–311
- prefixes for control names, 286
- Preserve statements, 133–134
- primary keys for databases, 366
- print events, 157–160
- printing Word documents, 403–404
- PrintWordDoc macro, 403–404
- prior selected cells, coloring, 375–376
- Project Explorer window, 26–27, 150
- prompts
input boxes, 94
message boxes, 93
- properties and Property Window, 27
accessing, 339
IntelliSense for, 71–73
module names, 41
Object Browser, 29
object model, 49, 51
UserForms, 273–274
- protecting
add-in code, 348
VBE, 43–44
- PtrSafe keyword, 381, 424
- Public scope
arrays, 130
UDFs, 238
- PublicArrayExample macro, 130
- Q**
- queries, database, 366
- QueryClose events, 307
- QueryTables
- for Access, 356–358
from web queries, 353–356
- question marks (?) for Immediate window, 261–262
- Quick Watch window, 263
- QuickBASIC language, 4
- quotes (“)
column references, 76
ranges, 77
VALUES clause, 370
- R**
- R1C1 references in formulas, 113–115
vs. A1, 115
toggling with A1, 116–118
- RAND function, 124, 243
- random numbers
lottery example, 110–111
volatility of, 124, 243
- ranges and Range object, 50, 75
continuously populated, 75–77
copying
to PowerPoint presentations, 432–433
to Word documents, 402–403
- with duplicates, 171–172
- identifying, 79–80
- last rows and columns, 80–81
- named, 119–120
- noncontinuously populated, 77
- OFFSET property, 78
- overview, 69–70
- RESIZE property, 78
- SpecialCells, 70–71
- start and end, 81–82
- step-by-step example, 82–84
- transferring to e-mail, 411–413
- readability, variables for, 57
- real-time charts, 314–315

recalculating
 calculate events for, 144
 Volatile functions, 124, 243

Record Macro dialog box, 18
 recording macros, 16–21
 records
 adding to Access, 419–423
 databases, 366

Recordset object, 367–368
 recurring tasks, 5
 RecycleBinEmpty macro, 382
 ReDim statements, 133–134
 references in formulas, 113–115
 A1 vs. R1C1, 115
 converting absolute and relative,
 121–122
 mixed, for filling empty cells, 118–119
 named ranges with, 119–120
 toggling between style views, 116–118

Refresh method, 52
 RefreshAll method, 232
 refreshing
 PivotCaches, 226
 PivotTables, 232
 QueryTables, 355–356

relational databases, 366
 relative references
 converting to absolute, 121–122
 named ranges with, 119–120

removing add-in list items, 349–350
 RenameCharts macro, 208
 renaming
 charts, 208
 modules, 41–42

repeating actions with loops. See loops
 repetitive tasks, 5
 Require Variable Declaration option, 61, 68
 Reset button, 255
 RESIZE property, 78
 reverse loops, 105–106
 Ribbon interface, 11, 13

right click events
 workbooks, 156
 worksheets, 142

rows
 coloring, 373–375
 deleting
 with duplicates, 161–167
 filtered dates, 378–379
 in SQL, 370–371

inserting
 on data changes, 172–173
 in databases, 369–370
 input boxes for, 94
 loops for, 106
 last, 80–81

RowSource property
 ComboBox controls, 292–293
 ListBox controls, 291

Run button, 255
 Run Macro button, 21
 running macros, 21–22
 runtime errors, 253

S

Same Size button, 306
 Save As dialog box, 341, 349–350
 save events, 158
 SaveCellValue macro, 361
 Saved property, 51
 scope
 arrays, 130
 constants, 63–64
 variables, 61–63

ScreenUpdating, 175
 searching
 loops for, 109
 in Object Browser, 30

Select Case structure, 91–92
 Select Data Source dialog box, 357
 Select method, 37

SELECT statement in SQL, 369
Select Table dialog box, 357–359, 426
SelectCaseExample macro, 92
SelectDataRange macro, 79
selected cells, coloring, 375–376
SelectedWorksheets macro, 133–134
selecting
 photographs, 308–309
 range of duplicates, 171–172
 worksheets, 107–108
selection change events
 workbooks, 155
 worksheets, 141–142
Selection object, 69
SelectSheet macro, 107–108
SelectUsedRange macro, 79
self-expiring workbooks, 382
Send to Back button, 306
SendEmail macro, 410–411
SendMail, 415
Set as Default Chart option, 200
SheetManager macro, 339, 346–347
SheetName UDF, 244
SheetPivotTableUpdate events, 156
Sheets collection, 69
ShellExecute function, 424
Shift+F9 keys, 263
shortcut keys for macros, 19, 22
Show Developer tab, 13–14
ShowAllItems macro, 231
ShowHidePivotChartFieldButtons macro,
 225–226
ShowModal property, 306
ShowSingleItem macro, 231
ShowUserForm1 macro, 306
ShowUserForm2 macro, 307
Single data type, 58
64-bit version, 381
size
 arrays, 129, 133–134
 UserForms, 308
slides in PowerPoint presentations
 copying chart sheets to, 433–435
 copying ranges to, 432–433
Solver add-in, 346
Sort_Separate_ClientName macro, 172–173
SpecialCells method, 70–71, 118
splash screens, 309–310
SQL. See Structured Query Language (SQL)
standard modules
 description, 28
 UDFs, 238
start of ranges, 81–82
static random numbers, 124
Static scope of arrays, 130
StaticRandom UDF, 243
Step statements, 105–106
stepping through code, 255–256
 Step Into button, 257–258
 Step Out button, 259
 Step Over button, 258–259
stock quotes QueryTables example, 353–356
Stop Recording toolbar, 20–21
strings and String data type
 description, 58
 extracting characters from, 241–242
Structured Query Language (SQL),
 356, 368
 DELETE statement, 370–371
 examples, 371–372
 INSERT statement, 369–370
 SELECT statement, 369
 UPDATE statement, 370
 uppercase for statements, 369
Sub statement, 36
SumAlongOneRow macro, 123–124
SumColor UDF, 240
SUMIF function, 239
summing
 lists, 122–124
 numbers in colored cells, 239–240
syntax errors, 252

T**tables**

Access. See Access application and tables
arrays as, 128–129

PivotTables. See PivotTables

TestComment UDF, 244–245

TestPublicArrayExample macro, 130

TestSheetCreate macro, 265–266

text files for external data, 359–361

TextBox controls

- adding, 276–277
- collections of, 326
- input filtering, 323–326
- overview, 288–290

TextBox1_KeyPress macro, 289, 323–324

TextExport macro, 360–361

32-bit version, 381

ThisWorkbook module, 28, 150, 158, 202

time

- declaring, 58–59
- fractional values with, 287
- querying, 361–364

Time function, 287

TimeAfterTime macro, 363

titles in message boxes, 93

Toggle Breakpoint button, 259–261

ToggleViews macro, 195–197

toolbars

- Control Toolbox, 186–187
- Debug. See Debug toolbar
- Forms, 143, 182–183
- Macro Recorder, 20
- Stop Recording, 21
- UserForms, 305–306
- VB, 12–13
- VBE, 33–34

Toolbox in VBE, 274–276

Top parameter for charts, 209

trapping errors, 264–266

triggers

for automatically run macros, 5–6
events. See events

True value in truth tables, 85–88

truth tables, 85–88

TryItPieChart macro, 209–211

two-dimensional arrays, 129

TwoDimensionalArray macro, 129

TxtGroup_KeyPress macro, 324

U

UBound function, 130, 132

UDFs. See user-defined functions (UDFs)

Ungroup button, 306

UnhideSheets macro, 104

unhiding worksheets, 104

unique lists from multiple columns, 167–170

UniqueList macro, 167–168

UniqueStoresToWorkbooks macro, 174–179

Unload method, 281

unloading UserForms, 282, 309–310

UPDATE statement in SQL, 370

UsedRange property, 79–80

user decisions, 92

- input boxes, 94

- message boxes, 92–93

user-defined functions (UDFs)

- anatomy, 238–239

- calling from macros, 245–246

- characteristics, 237–238

- conditional formatting, 244–245

- creating, 7

- description, 237

- extracting addresses from hyperlinks, 242

- extracting characters from strings, 241–242

- Insert Function dialog box, 246–248

- returning active worksheet and workbook names, 243–244

- step-by-step example, 248–249

- summing numbers, 239–240

- volatile, 243

- user interface for add-ins, 346–348
 - UserForm_Initialize macro
 - charts, 314
 - ComboBoxes, 292–294
 - labels, 287
 - ListBoxes, 291, 310–311, 313
 - TextBoxes, 325–326
 - UserForm size, 308–309
 - UserForm_QueryClose macro, 307
 - UserForms
 - add-ins, 346
 - Close button, 307–308
 - closing, 281–283
 - code, 280–281
 - controls. See controls
 - creating, 272–273
 - description, 271
 - designing, 273–274
 - hiding, 283
 - ListBox and ComboBox items
 - populating, 312–314
 - pre-sorting, 310–311
 - modal vs. modeless, 306–307
 - modules, 28
 - photographs, 308–309
 - real-time charts, 314–315
 - showing, 280
 - size, 308
 - step-by-step examples, 283–284, 315–319
 - toolbar, 305–306
 - unloading, 282, 309–310
- V**
- Val function, 289
 - Value Field Settings dialog box, 219–221
 - values, assigning to variables, 56
 - VALUES clause in INSERT statement, 370
 - variables
 - assigning values to, 56
 - data types
 - dates and time, 58–59
 - declaring, 59–61
 - overview, 57–58
 - declaring, 55–56, 59–61
 - names, 55–56
 - need for, 56–57
 - overview, 55–56
 - scope, 61–63
 - step-by-step example, 64–66
 - Watch window, 262–263
 - workbooks, 68
 - Variant data type
 - as default type, 59
 - description, 58
 - VB (Visual Basic) vs. VBA, 4
 - VBA overview, 3
 - benefits, 5–6
 - controlling non-Excel applications, 7–8
 - environment access, 11–15
 - history, 4
 - liabilities, 8
 - UDFs, 7
 - workbook look and feel, 7
 - VBAProject - Project Properties dialog box, 43
 - VBE. See Visual Basic Editor (VBE)
 - vbModal value, 306
 - versions
 - Office, 392
 - VBA, 11–12
 - View Code option, 138
 - visibility
 - arrays, 130
 - variables, 61–64
 - Visual Basic Editor icon, 31

Visual Basic Editor (VBE)
 description, 25
 exiting, 30
 getting into, 25–26
 locking and protecting, 43–44
 macros
 code, 36
 deleting, 39
 editing, 37–39
 locating, 33–35
 modules, 28
 deleting, 42–43
 inserting, 39–40
 renaming, 41–42
 Object Browser, 28–30
 step-by-step example, 30–31, 44–45
 toolbars, 33–34
 UserForms. See UserForms
 windows, 26–28

Visual Basic toolbar, 12–13
 volatile functions, 124, 243

W

Watch window, 262–263
 web queries, QueryTables from, 353–356
 WEEKDAY function, 88, 133
 WeekdayTest macro, 90
 WHERE clause
 DELETE statement, 370–371
 SELECT statement, 369
 While...Wend loops, 110
 Width parameter for charts, 209
 Windows API, 381–382
 With statements, 39
 WithEvents keyword, 324
 WithoutVariable macro, 56–57

WithVariable macro, 57
 Word application and documents
 activating, 399–401
 copying ranges to, 402–403
 creating, 402
 early binding, 392–394
 importing, 404–405
 opening, 400–401
 printing, 403–404
 step-by-step example, 405–408
 Workbook_Activate events, 154
 Workbook_BeforeClose events, 154, 347
 Workbook_BeforePrint events, 157
 Workbook_BeforeSave events, 158, 359
 Workbook_Deactivate events, 154
 workbook events
 code for, 151–152
 common, 153–158
 modules for, 149–151
 overview, 149
 step-by-step example, 158–160
 Workbook_NewSheet events, 156–157
 Workbook object, 50, 67
 Workbook_Open events
 ActiveX controls, 328
 add-ins, 347
 CheckBox controls, 329–330
 overview, 153–154
 PivotTables, 232
 Workbook_SheetActivate events, 157
 Workbook_SheetBeforeDoubleClick events, 155–156
 Workbook_SheetBeforeRightClick events, 156
 Workbook_SheetChange events, 154–155
 Workbook_SheetDeactivate events, 157–158
 Workbook_SheetSelectionChange events, 155

- workbooks
- closing, 105
 - events. See [workbook events](#)
 - look and feel, 7
 - modules, 28
 - opening, 107
 - self-expiring, 382
 - working with, 67–69
- [Workbooks collection](#), 53, 67–69
- [Worksheet_Activate events](#), 143
- [Worksheet_BeforeDoubleClick events](#), 142, 171–172
- [Worksheet_BeforeRightClick events](#), 142
- [Worksheet_Calculate events](#), 144
- [Worksheet_Change events](#), 140
 - cell change logs, 380
 - overview, 140
 - PivotCharts, 229
 - PivotTables, 231–232
 - summing numbers, 144–148
 - unique items, 169–170
- [Worksheet_Deactivate events](#), 144
- worksheet events, 137
 - common, 141–144
 - description, 137–138
 - enabling and disabling, 139–140
 - modules for, 138–139
 - step-by-step example, 144–148
 - [Worksheet_Change](#), 141
- [Worksheet_FollowHyperlink events](#), 142–143
- [Worksheet object](#), 50
- [Worksheet_PivotTableUpdate events](#), 144
- [Worksheet_SelectionChange events](#)
- coloring active elements, 373–375
- coloring cells, 376
- overview, 141–142
- worksheets
- adding, 68–69
 - adding embedded charts to, 202–204
 - closing, 104
 - creating, 109
 - e-mailing, 415
 - events. See [worksheet events](#)
 - modules, 28
 - selecting, 107–108
 - unhiding, 104
- [Worksheets collection](#), 52, 69
- [WorksheetTest1 macro](#), 68
- [WorksheetTest2 macro](#), 69

X

- .xla extension, 335
- .xlam extension, 335
- xlPath UDF, 239
- .xlsx extension, 176

Y

- years, entering, 59
- [YearSheets macro](#), 109

Z

- zero-based array numbering, 130–131
- [Zoom button](#), 306

Try Safari Books Online FREE for 15 days and take 15% off for up to 6 Months*

Gain unlimited subscription access to thousands of books and videos.



With Safari Books Online, learn without limits from thousands of technology, digital media and professional development books and videos from hundreds of leading publishers. With a monthly or annual unlimited access subscription, you get:

- Anytime, anywhere mobile access with Safari To Go apps for iPad, iPhone and Android
- Hundreds of expert-led instructional videos on today's hottest topics
- Sample code to help accelerate a wide variety of software projects
- Robust organizing features including favorites, highlights, tags, notes, mash-ups and more
- Rough Cuts pre-published manuscripts

START YOUR FREE TRIAL TODAY!

Visit: www.safaribooksonline.com/wrox

*Discount applies to new Safari Library subscribers only and is valid for the first 6 consecutive monthly billing cycles. Safari Library is not available in all countries.



An Imprint of  **WILEY**
Now you know.



Connect with Wrox.

Participate

Take an active role online by participating in our P2P forums @ p2p.wrox.com

Wrox Blox

Download short informational pieces and code to keep you up to date and out of trouble

Join the Community

Sign up for our free monthly newsletter at newsletter.wrox.com

Wrox.com

Browse the vast selection of Wrox titles, e-books, and blogs and find exactly what you need

User Group Program

Become a member and take advantage of all the benefits

Wrox on

Follow @wrox on Twitter and be in the know on the latest news in the world of Wrox

Wrox on

Join the Wrox Facebook page at facebook.com/wroxpress and get updates on new books and publications as well as upcoming programmer conferences and user group events

Contact Us.

We love feedback! Have a book idea? Need community support?
Let us know by e-mailing wrox-partnerwithus@wrox.com

WILEY END USER LICENSE AGREEMENT

Go to www.wiley.com/go/eula to access Wiley's ebook EULA.