

Excel® 2016

VBA and MACROS

Bill Jelen
Tracy Syrstad



Excel® 2016

VBA and MACROS

This book is part of Que's exciting new Content Update Program, which provides automatic content updates for major technology improvements!

- ▶ As Microsoft makes significant updates to Excel 2016, sections of this book will be updated or new sections will be added to match the updates to the software.
- ▶ The updates will be delivered to you via a free Web Edition of this book, which can be accessed with any Internet connection.
- ▶ This means your purchase is protected from immediately outdated information!

For more information on Que's Content Update program, see the inside back cover or go to

www.quepublishing.com/CUP.



**Content Update
Program**

*If you have additional questions, please email our
Customer Service department at informat@custhelp.com.*



Excel® 2016 VBA and Macros

Bill Jelen

Tracy Syrstad

que®

800 E. 96th Street
Indianapolis, Indiana 46240

Contents at a Glance

Introduction.....	1
1 Unleashing the Power of Excel with VBA	7
2 This Sounds Like BASIC, So Why Doesn't It Look Familiar?	33
3 Referring to Ranges	59
4 Looping and Flow Control	73
5 R1C1-Style Formulas	93
6 Creating and Manipulating Names in VBA	103
7 Event Programming	115
8 Arrays	131
9 Creating Classes and Collections	139
10 Userforms: An Introduction	157
11 Data Mining with Advanced Filter	177
12 Using VBA to Create Pivot Tables	211
13 Excel Power	251
14 Sample User-Defined Functions	283
15 Creating Charts	309
16 Data Visualizations and Conditional Formatting	333
17 Dashboarding with Sparklines in Excel 2016	355
18 Reading from and Writing to the Web	375
19 Text File Processing	391
20 Automating Word	405
21 Using Access as a Back End to Enhance Multiuser Access to Data	423
22 Advanced Userform Techniques	439
23 The Windows Application Programming Interface (API)	463
24 Handling Errors	473
25 Customizing the Ribbon to Run Macros	487
26 Creating Add-ins	509
27 An Introduction to Creating Office Add-ins	517
28 What's New in Excel 2016 and What's Changed	539
Index	545

Excel® 2016 VBA and Macros

Copyright © 2016 by Pearson Education, Inc.

All rights reserved. No part of this book shall be reproduced, stored in a retrieval system, or transmitted by any means, electronic, mechanical, photocopying, recording, or otherwise, without written permission from the publisher. No patent liability is assumed with respect to the use of the information contained herein. Although every precaution has been taken in the preparation of this book, the publisher and author assume no responsibility for errors or omissions. Nor is any liability assumed for damages resulting from the use of the information contained herein.

ISBN-13: 978-0-7897-5585-8

ISBN-10: 0-7897-5585-8

Library of Congress Control Number: 2015950785

Printed in the United States of America

First Printing: November 2015

Trademarks

All terms mentioned in this book that are known to be trademarks or service marks have been appropriately capitalized. Que Publishing cannot attest to the accuracy of this information. Use of a term in this book should not be regarded as affecting the validity of any trademark or service mark.

Warning and Disclaimer

Every effort has been made to make this book as complete and as accurate as possible, but no warranty or fitness is implied. The information provided is on an “as is” basis. The authors and the publisher shall have neither liability nor responsibility to any person or entity with respect to any loss or damages arising from the information contained in this book.

Special Sales

For information about buying this title in bulk quantities, or for special sales opportunities (which may include electronic versions; custom cover designs; and content particular to your business, training goals, marketing focus, or branding interests), please contact our corporate sales department at corpsales@pearsoned.com or (800) 382-3419.

For government sales inquiries, please contact governmentsales@pearsoned.com.

For questions about sales outside the U.S., please contact international@pearsoned.com.

Editor-in-Chief

Greg Wiegand

Acquisitions Editor

Joan Murray

Development Editor

Charlotte Kughen

Managing Editor

Sandra Schroeder

Project Editor

Mandie Frank

Copy Editor

Kitty Wilson

Indexer

Ken Johnson

Proofreader

Dan Knott

Technical Editor

Bob Umlas

Editorial Assistant

Cindy Teeters

Designer

Chuti Prasertsith

Compositor

Trina Wurst

Contents

Introduction	1
What Is in This Book?	1
Reducing the Learning Curve	1
Excel VBA Power	2
Techie Stuff Needed to Produce Applications	2
Does This Book Teach Excel?	2
The Future of VBA and Windows Versions of Excel	4
Versions of Excel	4
Differences for Mac Users	4
Special Elements and Typographical Conventions	5
Code Files	5
Next Steps	5
1 Unleashing the Power of Excel with VBA	7
The Power of Excel	7
Barriers to Entry	7
The Macro Recorder Doesn't Work!	7
No One on the Excel Team Is Focused on the Macro Recorder	8
Visual Basic Is Not Like BASIC	8
Good News: Climbing the Learning Curve Is Easy	9
Great News: Excel with VBA Is Worth the Effort	9
Knowing Your Tools: The Developer Tab	9
Understanding Which File Types Allow Macros	10
Macro Security	12
Adding a Trusted Location	12
Using Macro Settings to Enable Macros in Workbooks Outside Trusted Locations	13
Using Disable All Macros with Notification	14
Overview of Recording, Storing, and Running a Macro	14
Filling Out the Record Macro Dialog	15
Running a Macro	16
Creating a Macro Button on the Ribbon	16
Creating a Macro Button on the Quick Access Toolbar	17
Assigning a Macro to a Form Control, Text Box, or Shape	18
Understanding the VB Editor	19
VB Editor Settings	20
The Project Explorer	20
The Properties Window	21
Understanding Shortcomings of the Macro Recorder	21
Recording the Macro	23
Examining Code in the Programming Window	23
Running the Macro on Another Day Produces Undesired Results	25
Possible Solution: Use Relative References When Recording	26
Never Use AutoSum or Quick Analysis While Recording a Macro	30

Four Tips for Using the Macro Recorder.....	31
Next Steps.....	32
2 This Sounds Like BASIC, So Why Doesn't It Look Familiar?.....	33
I Can't Understand This Code.....	33
Understanding the Parts of VBA "Speech"	34
VBA Is Not Really Hard	37
VBA Help Files: Using F1 to Find Anything	38
Using Help Topics	38
Examining Recorded Macro Code: Using the VB Editor and Help	39
Optional Parameters	39
Defined Constants	40
Properties Can Return Objects	43
Using Debugging Tools to Figure Out Recorded Code	43
Stepping Through Code	43
More Debugging Options: Breakpoints.....	45
Backing Up or Moving Forward in Code	45
Not Stepping Through Each Line of Code.....	46
Querying Anything While Stepping Through Code	46
Using a Watch to Set a Breakpoint.....	49
Using a Watch on an Object	49
Object Browser: The Ultimate Reference	50
Seven Tips for Cleaning Up Recorded Code	51
Tip 1: Don't Select Anything.....	51
Tip 2: Use <code>Cells(2, 5)</code> Because It's More Convenient Than <code>Range("E2")</code>	52
Tip 3: Use More Reliable Ways to Find the Last Row.....	52
Tip 4: Use Variables to Avoid Hard-Coding Rows and Formulas.....	53
Tip 5: Use R1C1 Formulas That Make Your Life Easier	54
Tip 6: Copy and Paste in a Single Statement.....	54
Tip 7: Use <code>With...End With</code> to Perform Multiple Actions	54
Next Steps.....	57
3 Referring to Ranges.....	59
The Range Object.....	59
Syntax for Specifying a Range.....	60
Named Ranges.....	60
Shortcut for Referencing Ranges.....	60
Referencing Ranges in Other Sheets	61
Referencing a Range Relative to Another Range.....	61
Using the <code>Cells</code> Property to Select a Range.....	62
Using the <code>Offset</code> Property to Refer to a Range.....	63
Using the <code>Resize</code> Property to Change the Size of a Range.....	65
Using the <code>Columns</code> and <code>Rows</code> Properties to Specify a Range.....	66
Using the <code>Union</code> Method to Join Multiple Ranges.....	66

Using the <code>Intersect</code> Method to Create a New Range from Overlapping Ranges.....	67
Using the <code>IsEmpty</code> Function to Check Whether a Cell Is Empty.....	67
Using the <code>CurrentRegion</code> Property to Select a Data Range.....	68
Using the <code>Areas</code> Collection to Return a Noncontiguous Range.....	70
Referencing Tables.....	71
Next Steps.....	72
4 Looping and Flow Control.....	73
For...Next Loops.....	73
Using Variables in the For Statement.....	75
Variations on the For...Next Loop.....	76
Exiting a Loop Early After a Condition Is Met.....	77
Nesting One Loop Inside Another Loop.....	78
Do Loops.....	78
Using the While or Until Clause in Do Loops.....	81
The VBA Loop: For Each.....	82
Object Variables.....	83
Flow Control: Using If...Then...Else and Select Case.....	86
Basic Flow Control: If...Then...Else.....	86
Using Select Case...End Select for Multiple Conditions.....	88
Next Steps.....	91
5 R1C1-Style Formulas.....	93
Referring to Cells: A1 Versus R1C1 References.....	93
Toggling to R1C1-Style References.....	94
Witnessing the Miracle of Excel Formulas.....	95
Entering a Formula Once and Copying 1,000 Times.....	95
The Secret: It's Not That Amazing.....	96
Understanding the R1C1 Reference Style.....	97
Using R1C1 with Relative References.....	97
Using R1C1 with Absolute References.....	98
Using R1C1 with Mixed References.....	98
Referring to Entire Columns or Rows with R1C1 Style.....	99
Replacing Many A1 Formulas with a Single R1C1 Formula.....	99
Remembering Column Numbers Associated with Column Letters.....	101
Using R1C1 Formulas with Array Formulas.....	101
Next Steps.....	102
6 Creating and Manipulating Names in VBA.....	103
Global Versus Local Names.....	103
Adding Names.....	104
Deleting Names.....	105
Adding Comments.....	106
Types of Names.....	106
Formulas.....	106

Strings.....	107
Numbers.....	108
Tables.....	109
Using Arrays in Names.....	109
Reserved Names.....	110
Hiding Names.....	111
Checking for the Existence of a Name.....	111
Next Steps.....	114
7 Event Programming.....	115
Levels of Events.....	115
Using Events.....	116
Event Parameters.....	116
Enabling Events.....	117
Workbook Events.....	117
Workbook-Level Sheet and Chart Events.....	119
Worksheet Events.....	120
Chart Events.....	123
Embedded Charts.....	123
Embedded Chart and Chart Sheet Events.....	124
Application-Level Events.....	125
Next Steps.....	130
8 Arrays.....	131
Declaring an Array.....	131
Declaring a Multidimensional Array.....	132
Filling an Array.....	133
Retrieving Data from an Array.....	134
Using Arrays to Speed Up Code.....	135
Using Dynamic Arrays.....	136
Passing an Array.....	137
Next Steps.....	138
9 Creating Classes and Collections.....	139
Inserting a Class Module.....	139
Trapping Application and Embedded Chart Events.....	140
Application Events.....	140
Embedded Chart Events.....	141
Creating a Custom Object.....	143
Using a Custom Object.....	145
Using Collections.....	145
Creating a Collection.....	146
Creating a Collection in a Standard Module.....	146
Creating a Collection in a Class Module.....	148

Using Dictionaries	150
Using User-Defined Types to Create Custom Properties	153
Next Steps	156
10 Userforms: An Introduction	157
Input Boxes	157
Message Boxes	158
Creating a Userform	158
Calling and Hiding a Userform	159
Programming Userforms	160
Userform Events	160
Programming Controls	162
Using Basic Form Controls	163
Using Labels, Text Boxes, and Command Buttons	163
Deciding Whether to Use List Boxes or Combo Boxes in Forms	165
Adding Option Buttons to a Userform	167
Adding Graphics to a Userform	169
Using a Spin Button on a Userform	170
Using the <code>MultiPage</code> Control to Combine Forms	171
Verifying Field Entry	174
Illegal Window Closing	174
Getting a Filename	175
Next Steps	176
11 Data Mining with Advanced Filter	177
Replacing a Loop with <code>AutoFilter</code>	177
Using <code>AutoFilter</code> Techniques	180
Selecting Visible Cells Only	183
Advanced Filter—Easier in VBA Than in Excel	184
Using the Excel Interface to Build an Advanced Filter	185
Using Advanced Filter to Extract a Unique List of Values	186
Extracting a Unique List of Values with the User Interface	186
Extracting a Unique List of Values with VBA Code	187
Getting Unique Combinations of Two or More Fields	191
Using Advanced Filter with Criteria Ranges	192
Joining Multiple Criteria with a Logical OR	193
Joining Two Criteria with a Logical AND	194
Other Slightly Complex Criteria Ranges	194
The Most Complex Criteria: Replacing the List of Values with a Condition Created as the Result of a Formula	194
Using Filter in Place in Advanced Filter	201
Catching No Records When Using a Filter in Place	202
Showing All Records After Running a Filter in Place	202
The Real Workhorse: <code>xlFilterCopy</code> with All Records Rather Than Unique Records Only	203
Copying All Columns	203

Copying a Subset of Columns and Reordering	204
Excel in Practice: Turning Off a Few Drop-downs in the AutoFilter	209
Next Steps	210
12 Using VBA to Create Pivot Tables	211
Understanding How Pivot Tables Evolved Over Various Excel Versions	211
While Building a Pivot Table in Excel VBA	212
Defining the Pivot Cache	212
Creating and Configuring the Pivot Table	213
Adding Fields to the Data Area	214
Learning Why You Cannot Move or Change Part of a Pivot Report	216
Determining the Size of a Finished Pivot Table to Convert the Pivot Table to Values	217
Using Advanced Pivot Table Features	219
Using Multiple Value Fields	220
Grouping Daily Dates to Months, Quarters, or Years	221
Changing the Calculation to Show Percentages	222
Eliminating Blank Cells in the Values Area	225
Controlling the Sort Order with AutoSort	225
Replicating the Report for Every Product	225
Filtering a Data Set	228
Manually Filtering Two or More Items in a Pivot Field	228
Using the Conceptual Filters	229
Using the Search Filter	233
Setting Up Slicers to Filter a Pivot Table	235
Setting Up a Timeline to Filter an Excel 2016 Pivot Table	239
Using the Data Model in Excel 2016	242
Adding Both Tables to the Data Model	242
Creating a Relationship Between the Two Tables	243
Defining the PivotCache and Building the Pivot Table	243
Adding Model Fields to the Pivot Table	244
Adding Numeric Fields to the Values Area	244
Putting It All Together	245
Using Other Pivot Table Features	247
Calculated Data Fields	247
Calculated Items	247
Using ShowDetail to Filter a Record Set	248
Changing the Layout from the Design Tab	248
Settings for the Report Layout	248
Suppressing Subtotals for Multiple Row Fields	249
Next Steps	250
13 Excel Power	251
File Operations	251
Listing Files in a Directory	251
Importing and Deleting a CSV File	254
Reading a Text File into Memory and Parsing	254

Combining and Separating Workbooks	255
Separating Worksheets into Workbooks.....	255
Combining Workbooks.....	256
Filtering and Copying Data to Separate Worksheets.....	257
Copying Data to Separate Worksheets Without Using Filter.....	258
Exporting Data to an XML File.....	259
Working with Cell Comments	260
Resizing Comments.....	260
Placing a Chart in a Comment.....	261
Selecting Cells	263
Using Conditional Formatting to Highlight the Selected Cell.....	263
Highlighting the Selected Cell Without Using Conditional Formatting.....	264
Selecting/Deselecting Noncontiguous Cells.....	265
Creating a Hidden Log File.....	267
Techniques for VBA Pros.....	268
Creating an Excel State Class Module	268
Drilling-Down a Pivot Table.....	270
Filtering an OLAP Pivot Table by a List of Items.....	271
Creating a Custom Sort Order	273
Creating a Cell Progress Indicator	274
Using a Protected Password Box	275
Changing Case.....	277
Selecting with SpecialCells.....	279
Resetting a Table's Format.....	279
Cool Applications.....	280
Getting Historical Stock/Fund Quotes.....	280
Using VBA Extensibility to Add Code to New Workbooks.....	281
Next Steps.....	282
14 Sample User-Defined Functions.....	283
Creating User-Defined Functions.....	283
Sharing UDFs.....	286
Useful Custom Excel Functions.....	286
Setting the Current Workbook's Name in a Cell.....	286
Setting the Current Workbook's Name and File Path in a Cell.....	287
Checking Whether a Workbook Is Open.....	287
Checking Whether a Sheet in an Open Workbook Exists.....	287
Counting the Number of Workbooks in a Directory.....	288
Retrieving the User ID.....	289
Retrieving Date and Time of Last Save.....	291
Retrieving Permanent Date and Time.....	291
Validating an Email Address.....	292
Summing Cells Based on Interior Color.....	293
Counting Unique Values.....	294
Removing Duplicates from a Range.....	295

Finding the First Nonzero-Length Cell in a Range	296
Substituting Multiple Characters	297
Retrieving Numbers from Mixed Text	298
Converting Week Number into Date	299
Extracting a Single Element from a Delimited String	300
Sorting and Concatenating	300
Sorting Numeric and Alpha Characters	302
Searching for a String Within Text	303
Reversing the Contents of a Cell	304
Returning the Addresses of Duplicate Max Values	304
Returning a Hyperlink Address	305
Returning the Column Letter of a Cell Address	306
Using Static Random	306
Using <code>Select Case</code> on a Worksheet	307
Next Steps	308
15 Creating Charts	309
Contrasting the Good and Bad VBA to Create Charts	309
Planning for More Charts to Break	310
Using <code>.AddChart2</code> to Create a Chart	311
Understanding Chart Styles	312
Formatting a Chart	315
Referring to a Specific Chart	315
Specifying a Chart Title	316
Applying a Chart Color	317
Filtering a Chart	318
Using <code>SetElement</code> to Emulate Changes from the Plus Icon	319
Using the <code>Format</code> Method to Micromanage Formatting Options	324
Changing an Object's Fill	325
Formatting Line Settings	327
Creating a Combo Chart	327
Exporting a Chart as a Graphic	330
Considering Backward Compatibility	331
Next Steps	331
16 Data Visualizations and Conditional Formatting	333
VBA Methods and Properties for Data Visualizations	334
Adding Data Bars to a Range	335
Adding Color Scales to a Range	339
Adding Icon Sets to a Range	341
Specifying an Icon Set	341
Specifying Ranges for Each Icon	343
Using Visualization Tricks	343
Creating an Icon Set for a Subset of a Range	344
Using Two Colors of Data Bars in a Range	345

Using Other Conditional Formatting Methods	347
Formatting Cells That Are Above or Below Average	348
Formatting Cells in the Top 10 or Bottom 5	348
Formatting Unique or Duplicate Cells	349
Formatting Cells Based on Their Value	350
Formatting Cells That Contain Text	351
Formatting Cells That Contain Dates	351
Formatting Cells That Contain Blanks or Errors	351
Using a Formula to Determine Which Cells to Format	352
Using the New <code>NumberFormat</code> Property	353
Next Steps	354
17 Dashboarding with Sparklines in Excel 2016	355
Creating Sparklines	356
Scaling Sparklines	357
Formatting Sparklines	361
Using Theme Colors	361
Using RGB Colors	364
Formatting Sparkline Elements	365
Formatting Win/Loss Charts	368
Creating a Dashboard	369
Observations About Sparklines	369
Creating Hundreds of Individual Sparklines in a Dashboard	370
Next Steps	374
18 Reading from and Writing to the Web	375
Getting Data from the Web	375
Building Multiple Queries with VBA	377
Finding Results from Retrieved Data	378
Putting It All Together	379
Examples of Scraping Websites Using Web Queries	380
Using <code>Application.OnTime</code> to Periodically Analyze Data	381
Using Ready Mode for Scheduled Procedures	381
Specifying a Window of Time for an Update	382
Canceling a Previously Scheduled Macro	382
Closing Excel Cancels All Pending Scheduled Macros	383
Scheduling a Macro to Run <i>x</i> Minutes in the Future	383
Scheduling a Verbal Reminder	383
Scheduling a Macro to Run Every Two Minutes	384
Publishing Data to a Web Page	385
Using VBA to Create Custom Web Pages	386
Using Excel as a Content Management System	387
Bonus: FTP from Excel	389
Next Steps	390

19 Text File Processing	391
Importing from Text Files	391
Importing Text Files with Fewer Than 1,048,576 Rows	391
Dealing with Text Files with More Than 1,048,576 Rows	398
Writing Text Files	402
Next Steps	403
20 Automating Word	405
Using Early Binding to Reference a Word Object	406
Using Late Binding to Reference a Word Object	408
Using the New Keyword to Reference a Word Application	409
Using the CreateObject Function to Create a New Instance of an Object	409
Using the GetObject Function to Reference an Existing Instance of Word	410
Using Constant Values	411
Using the Watches Window to Retrieve the Real Value of a Constant	411
Using the Object Browser to Retrieve the Real Value of a Constant	412
Understanding Word's Objects	413
The Document Object	413
The Selection Object	415
The Range Object	416
Bookmarks	419
Controlling Form Fields in Word	420
Next Steps	422
21 Using Access as a Back End to Enhance Multiuser Access to Data	423
ADO Versus DAOs	424
The Tools of ADO	426
Adding a Record to a Database	427
Retrieving Records from a Database	429
Updating an Existing Record	431
Deleting Records via ADO	433
Summarizing Records via ADO	433
Other Utilities via ADO	434
Checking for the Existence of Tables	434
Checking for the Existence of a Field	435
Adding a Table On the Fly	436
Adding a Field On the Fly	436
SQL Server Examples	437
Next Steps	438
22 Advanced Userform Techniques	439
Using the UserForm Toolbar in the Design of Controls on Userforms	439
More Userform Controls	440
Checkbox Controls	440

Controls and Collections	447
Modeless Userforms	449
Using Hyperlinks in Userforms	449
Adding Controls at Runtime	450
Resizing the Userform On the Fly	452
Adding a Control On the Fly	452
Sizing On the Fly	452
Adding Other Controls	453
Adding an Image On the Fly	453
Putting It All Together	454
Adding Help to a Userform	456
Showing Accelerator Keys	456
Adding Control Tip Text	457
Creating the Tab Order	457
Coloring the Active Control	457
Creating Transparent Forms	460
Next Steps	461
23 The Windows Application Programming Interface (API)	463
Understanding an API Declaration	464
Using an API Declaration	465
Making 32-Bit- and 64-Bit-Compatible API Declarations	465
API Function Examples	467
Retrieving the Computer Name	467
Checking Whether an Excel File Is Open on a Network	467
Retrieving Display-Resolution Information	468
Customizing the About Dialog	469
Disabling the X for Closing a Userform	470
Creating a Running Timer	471
Playing Sounds	472
Next Steps	472
24 Handling Errors	473
What Happens When an Error Occurs?	473
A Misleading Debug Error in Userform Code	475
Basic Error Handling with the On Error GoTo Syntax	477
Generic Error Handlers	478
Handling Errors by Choosing to Ignore Them	479
Suppressing Excel Warnings	481
Encountering Errors on Purpose	481
Training Your Clients	481
Errors While Developing Versus Errors Months Later	482
Runtime Error 9: Subscript Out of Range	482
Runtime Error 1004: Method Range of Object Global Failed	483
The Ills of Protecting Code	484

More Problems with Passwords	485
Errors Caused by Different Versions	486
Next Steps	486
25 Customizing the Ribbon to Run Macros	487
Where to Add Code: The customui Folder and File	488
Creating a Tab and a Group	489
Adding a Control to a Ribbon	490
Accessing the File Structure	496
Understanding the RELS File	496
Renaming an Excel File and Opening a Workbook	497
Using Images on Buttons	497
Using Microsoft Office Icons on a Ribbon	498
Adding Custom Icon Images to a Ribbon	499
Troubleshooting Error Messages	500
The Attribute “ Attribute Name ” on the Element “ customui Ribbon ” Is Not Defined in the DTD/Schema	500
Illegal Qualified Name Character	501
Element “ customui Tag Name ” Is Unexpected According to Content Model of Parent Element “ customui Tag Name ”	501
Found a Problem with Some Content	502
Wrong Number of Arguments or Invalid Property Assignment	503
Invalid File Format or File Extension	503
Nothing Happens	503
Other Ways to Run a Macro	504
Using a Keyboard Shortcut to Run a Macro	504
Attaching a Macro to a Command Button	504
Attaching a Macro to a Shape	505
Attaching a Macro to an ActiveX Control	506
Running a Macro from a Hyperlink	507
Next Steps	508
26 Creating Add-ins	509
Characteristics of Standard Add-ins	509
Converting an Excel Workbook to an Add-in	510
Using Save As to Convert a File to an Add-in	511
Using the VB Editor to Convert a File to an Add-in	512
Having a Client Install an Add-in	512
Closing Add-ins	514
Removing Add-ins	514
Using a Hidden Workbook as an Alternative to an Add-in	515
Next Steps	516
27 An Introduction to Creating Office Add-ins	517
Creating Your First Office Add-in—Hello World	517
Adding Interactivity to an Office Add-in	521

A Basic Introduction to HTML	524
Using Tags.....	524
Adding Buttons.....	524
Using CSS Files	525
Using XML to Define an Office Add-in	525
Using JavaScript to Add Interactivity to an Office Add-in	526
The Structure of a Function.....	526
Variables.....	527
Strings.....	528
Arrays.....	528
JavaScript for Loops	529
How to Do an if Statement in JavaScript.....	530
How to Do a Select . . Case Statement in JavaScript	530
How to Do a For each . . next Statement in JavaScript.....	532
Mathematical, Logical, and Assignment Operators.....	532
Math Functions in JavaScript.....	534
Writing to the Content Pane or Task Pane	535
JavaScript Changes for Working in an Office Add-in	535
Napa Office 365 Development Tools	536
Next Steps.....	537
28 What's New in Excel 2016 and What's Changed	539
If It Has Changed in the Front End, It Has Changed in VBA	539
The Ribbon	539
Single Document Interface (SDI).....	540
Quick Analysis Tool.....	541
Charts.....	541
Pivot Tables.....	541
Slicers.....	541
SmartArt	542
Learning the New Objects and Methods.....	542
Compatibility Mode	542
Using the Version Property	543
Using the Excel8CompatibilityMode Property.....	543
Next Steps.....	544
Index.....	545

About the Authors

Bill Jelen, Excel MVP and the host of MrExcel.com, has been using spreadsheets since 1985, and he launched the MrExcel.com website in 1998. Bill was a regular guest on *Call for Help* with Leo Laporte and has produced more than 1,900 episodes of his daily video podcast, *Learn Excel from MrExcel*. He is the author of 44 books about Microsoft Excel and writes the monthly Excel column for *Strategic Finance* magazine. Before founding MrExcel.com, Bill Jelen spent 12 years in the trenches—working as a financial analyst for finance, marketing, accounting, and operations departments of a \$500 million public company. He lives in Merritt Island, Florida, with his wife, Mary Ellen.

Tracy Syrstad is a Microsoft Excel developer and author of eight Excel books. She has been helping people with Microsoft Office issues since 1997, when she discovered free online forums where anyone could ask and answer questions. Tracy found out she enjoyed teaching others new skills, and when she began working as a developer, she was able to integrate the fun of teaching with one-on-one online desktop sharing sessions. Tracy lives on acreage in eastern South Dakota with her husband, one dog, two cats, one horse (two, hopefully soon), and a variety of wild foxes, squirrels, and rabbits.

Dedications

For Robert K. Jelen
—Bill Jelen

For Marlee Jo Jacobson
—Tracy Syrstad

Acknowledgments

Thanks to Tracy Syrstad for being a great coauthor.

Bob Umlas is the smartest Excel guy I know and is an awesome technical editor. At Pearson, Joan Murray is an excellent acquisitions editor.

Along the way, I've learned a lot about VBA programming from the awesome community at the MrExcel.com message board. VoG, Richard Schollar, and Jon von der Heyden all stand out as having contributed posts that led to ideas in this book. Thanks to Pam Gensel for Excel macro lesson #1. Mala Singh taught me about creating charts in VBA, and Oliver Holloway brought me up to speed with accessing SQL Server. Scott Ruble and Robin Wakefield at Microsoft helped with the charting chapter.

My family was incredibly supportive during this time. Thanks to Mary Ellen Jelen, Robert F. Jelen, and Robert K. Jelen.

—Bill

Juan Pablo Gonzalez Ruiz and Zack Barresse are great programmers, and I really appreciate their time and patience showing me new ways to write better programs. Chris “Smitty” Smith has really helped me sharpen my business acumen.

Thank you to all the moderators at the MrExcel forum who keep the board organized, despite the best efforts of the spammers.

Programming is a constant learning experience, and I really appreciate the clients who have encouraged me to program outside my comfort zone so that my skills and knowledge have expanded.

And last, but not least, thanks to Bill Jelen. His site, MrExcel.com, is a place where thousands come for help. It's also a place where I, and others like me, have an opportunity to learn from and assist others.

—Tracy

We Want to Hear from You!

As the reader of this book, *you* are our most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

We welcome your comments. You can email or write to let us know what you did or didn't like about this book—as well as what we can do to make our books better.

Please note that we cannot help you with technical problems related to the topic of this book.

When you write, please be sure to include this book's title and author as well as your name and email address. We will carefully review your comments and share them with the author and editors who worked on the book.

Email: feedback@quepublishing.com

Mail: Que Publishing
 ATTN: Reader Feedback
 800 East 96th Street
 Indianapolis, IN 46240 USA

Reader Services

Visit our website and register this book at quepublishing.com/register for convenient access to any updates, downloads, or errata that might be available for this book.

As corporate IT departments have found themselves with long backlogs of requests, Excel users have discovered that they can produce the reports needed to run their businesses themselves using the macro language *Visual Basic for Applications* (VBA). VBA enables you to achieve tremendous efficiencies in your day-to-day use of Excel. VBA helps you figure out how to import data and produce reports in Excel so that you don't have to wait for the IT department to help you.

What Is in This Book?

You have taken the right step by purchasing this book. We can help you reduce the learning curve so that you can write your own VBA macros and put an end to the burden of generating reports manually.

Reducing the Learning Curve

This Introduction provides a case study about the power of macros. Chapter 1, “Unleashing the Power of Excel with VBA,” introduces the tools and confirms what you probably already know: The macro recorder does not work reliably. Chapter 2, “This Sounds Like BASIC, So Why Doesn't It Look Familiar?” helps you understand the crazy syntax of VBA. Chapter 3, “Referring to Ranges,” cracks the code on how to work efficiently with ranges and cells.

Chapter 4, “Looping and Flow Control,” covers the power of looping using VBA. The case study in this chapter demonstrates creating a program to produce a department report and then wrapping that report routine in a loop to produce 46 reports.

Chapter 5, “R1C1-Style Formulas,” covers, obviously, R1C1-style formulas. Chapter 6, “Creating and Manipulate Names in VBA,” covers names. Chapter 7, “Event Programming,” includes some great tricks that use event programming. Chapters 8, “Arrays,”

INTRODUCTION

IN THIS INTRODUCTION

What Is in This Book?.....	1
The Future of VBA and Windows Versions of Excel	4
Special Elements and Typographical Conventions	5
Code Files	5
Next Steps	5



and 9, “Creating Classes and Collections,” cover arrays, classes, and collections. Chapter 10, “Userforms: An Introduction,” introduces custom dialog boxes that you can use to collect information from a human using Excel.

Excel VBA Power

Chapters 11, “Data Mining with Advanced Filter,” and 12, “Using VBA to Create Pivot Tables,” provide an in-depth look at Filter, Advanced Filter, and pivot tables. Report automation tools rely heavily on these concepts. Chapters 13, “Excel Power,” and 14, “Sample User-Defined Functions,” include dozens of code samples designed to exhibit the power of Excel VBA and custom functions.

Chapters 15, “Creating Charts,” through 20, “Automating Word,” handle charting, data visualizations, web queries, sparklines, and automating Word.

Techie Stuff Needed to Produce Applications

Chapter 21, “Using Access as a Back End to Enhance Multiuser Access to Data,” handles reading and writing to Access databases and SQL Server. The techniques for using Access databases enable you to build an application with the multiuser features of Access while keeping the friendly front end of Excel.

Chapter 22, “Advanced Userform Techniques,” shows you how to go further with userforms. Chapter 23, “The Windows Application Programming Interface (API),” teaches some tricky ways to achieve tasks using the Windows API. Chapters 24, “Handling Errors,” through 26, “Creating Add-ins,” deal with error handling, custom menus, and add-ins. Chapter 27, “An Introduction to Creating Office Add-Ins,” provides a brief introduction to building your own JavaScript application within Excel. Chapter 28, “What’s New in Excel 2016 and What’s Changed,” summarizes the changes in Excel 2016.

Does This Book Teach Excel?

Microsoft believes that the ordinary Office user touches only 10% of the features in Office. We realize that everyone reading this book is above average, and MrExcel.com has a pretty smart audience. Even so, a poll of 8,000 MrExcel.com readers showed that only 42% of smarter-than-average users are using any 1 of the top 10 power features in Excel.

I regularly present a Power Excel seminar for accountants. These are hard-core Excelers who use Excel 30 to 40 hours every week. Even so, two things come out in every seminar. First, half of the audience gasps when they see how quickly you can do tasks with a particular feature, such as automatic subtotals or pivot tables. Second, someone in the audience routinely trumps me. For example, someone asks a question, I answer, and someone in the second row raises a hand to give a better answer.

The point? You and I both know a lot about Excel. However, I assume that in any given chapter, maybe 58% of the people have not used pivot tables before and maybe even fewer have used the Top 10 Filter feature of pivot tables. With this in mind, before I show how to

automate something in VBA, I briefly cover how to do the same task in the Excel interface. This book does not teach you how to make pivot tables, but it does alert you when you might need to explore a topic and learn more about it elsewhere.

CASE STUDY: MONTHLY ACCOUNTING REPORTS

This is a true story. Valerie is a business analyst in the accounting department of a medium-size corporation. Her company recently installed an overbudget \$16 million enterprise resource planning (ERP) system. As the project ground to a close, there were no resources left in the IT budget to produce the monthly report that this corporation used to summarize each department.

However, Valerie had been close enough to the implementation to think of a way to produce the report herself. She understood that she could export general ledger data from the ERP system to a text file with comma-separated values. Using Excel, Valerie was able to import the general ledger data from the ERP system into Excel.

Creating the report was not easy. As in many other companies, there were exceptions in the data. Valerie knew that certain accounts in one particular cost center needed to be reclassified as expenses. She knew that other accounts needed to be excluded from the report entirely. Working carefully in Excel, Valerie made these adjustments. She created one pivot table to produce the first summary section of the report. She cut the pivot table results and pasted them into a blank worksheet. Then she created a new pivot table report for the second section of the summary. After about three hours, she had imported the data, produced five pivot tables, arranged them in a summary, and neatly formatted the report in color.

Becoming the Hero

Valerie handed the report to her manager. The manager had just heard from the IT department that it would be months before they could get around to producing “that convoluted report.” When Valerie created the Excel report, she became the instant hero of the day. In three hours, Valerie had managed to do the impossible. Valerie was on cloud nine after a well-deserved “atta-girl.”

More Cheers

The next day, Valerie’s manager attended the monthly department meeting. When the department managers started complaining that they could not get the report from the ERP system, this manager pulled out his department’s report and placed it on the table. The other managers were amazed. How was he able to produce this report? Everyone was relieved to hear that someone had cracked the code. The company president asked Valerie’s manager if he could have the report produced for each department.

Cheers Turn to Dread

You can probably see what’s coming. This particular company had 46 departments. That means 46 one-page summaries had to be produced once a month. Each report required importing data from the ERP system, backing out certain accounts, producing five pivot tables, and then formatting the reports in color. It had taken Valerie three hours to produce the first report, but after she got into the swing of things, she could produce the 46 reports in 40 hours. Even after she reduced her time per report, though, this is horrible. Valerie had a job to do before she became responsible for spending 40 hours a month producing these reports in Excel.

VBA to the Rescue

Valerie found my company, MrExcel Consulting, and explained her situation. In the course of about a week, I was able to produce a series of macros in Visual Basic that did all the mundane tasks. For example, the macros imported the data, backed out certain accounts, made five pivot tables, and applied the color formatting. From start to finish, the entire 40-hour manual process was reduced to two button clicks and about 4 minutes.

Right now, either you or someone in your company is probably stuck doing manual tasks in Excel that can be automated with VBA. I am confident that I can walk into any company that has 20 or more Excel users and find a case just as amazing as Valerie's.

The Future of VBA and Windows Versions of Excel

Several years ago, there were many rumblings that Microsoft might stop supporting VBA. There is now plenty of evidence to indicate that VBA will be around in Windows versions of Excel through 2036. When VBA was removed from the Mac version of Excel 2008, a huge outcry from customers led to its being included in the next Mac version of Excel.

XLM macros were replaced by VBA in 1993, and 23 years later, they are still supported. Microsoft is making strides toward providing a JavaScript alternative to VBA, but it appears that Excel will support VBA for about another 23 years.

Versions of Excel

This fifth edition of *VBA and Macros* is designed to work with Excel 2016. The previous editions of this book covered code for Excel 97 through Excel 2013. In 80% of the chapters, the code for Excel 2016 is identical to the code in previous versions. However, there are exceptions. For example, the new AutoGroup functionality in pivot tables adds new options that were not available in Excel 2013.

Differences for Mac Users

Although Excel for Windows and Excel for the Mac are similar in terms of user interface, there are a number of differences when you compare the VBA environment. Certainly, nothing in Chapter 23 that uses the Windows API will work on the Mac. That said, the overall concepts discussed in this book apply to the Mac. You can find a general list of differences as they apply to the Mac at <http://www.mrexcel.com/macvba.html>. Development in VBA for Mac Excel 2016 is far more difficult than in Windows, with only rudimentary VBA editing tools. Microsoft actually recommends that you write all of your VBA in Excel 2016 for Windows and then use that VBA on the Mac.

Special Elements and Typographical Conventions

The following typographical conventions are used in this book:

- *Italic*—Indicates new terms when they are defined, special emphasis, non-English words or phrases, and letters or words used as words.
- `Monospace`—Indicates parts of VBA code, such as object or method names.
- **Bold monospace**—Indicates user input.

In addition to these typographical conventions, there are several special elements. Each chapter has at least one case study that presents a real-world solution to common problems. The case study also demonstrates practical applications of topics discussed in the chapter.

In addition to the case studies, you will see Notes, Tips, and Cautions.

NOTE

Notes provide additional information outside the main thread of the chapter discussion that might be useful for you to know.

TIP

Tips provide quick workarounds and time-saving techniques to help you work more efficiently.

CAUTION

Cautions warn about potential pitfalls you might encounter. Pay attention to the Cautions; they alert you to problems that might otherwise cause you hours of frustration.

Code Files

As a thank-you for buying this book, we have put together a set of 50 Excel workbooks that demonstrate the concepts included in this book. This set of files includes all the code from the book, sample data, additional notes from the authors, and 25 bonus macros. To download the code files, visit this book's web page at <http://www.quepublishing.com> or <http://www.mrexcel.com/getcode2016.html>.

Next Steps

Chapter 1 introduces the editing tools of the Visual Basic environment and shows why using the macro recorder is not an effective way to write VBA macro code.

This page intentionally left blank

Unleashing the Power of Excel with VBA

1

The Power of Excel

Visual Basic for Applications (VBA) combined with Microsoft Excel is probably the most powerful tool available to you. VBA is sitting on the desktops of 750 million users of Microsoft Office, and most have never figured out how to harness the power of VBA in Excel. Using VBA, you can speed the production of any task in Excel. If you regularly use Excel to produce a series of monthly charts, for example, you can have VBA do that task for you in a matter of seconds.

Barriers to Entry

There are two barriers to learning successful VBA programming. First, Excel's macro recorder is flawed and does not produce workable code for you to use as a model. Second, for many who learned a programming language such as BASIC, the syntax of VBA is horribly frustrating.

The Macro Recorder Doesn't Work!

Microsoft began to dominate the spreadsheet market in the mid-1990s. Although it was wildly successful in building a powerful spreadsheet program to which any Lotus 1-2-3 user could easily transition, the macro language was just too different. Anyone proficient in recording Lotus 1-2-3 macros who tried recording a few macros in Excel most likely failed. Although the Microsoft VBA programming language is much more powerful than the Lotus 1-2-3 macro language, the fundamental flaw is that the macro recorder does not work when you use the default settings.

IN THIS CHAPTER

The Power of Excel	7
Barriers to Entry.....	7
Knowing Your Tools: The Developer Tab.....	9
Understanding Which File Types Allow Macros.....	10
Macro Security	12
Overview of Recording, Storing, and Running a Macro	14
Running a Macro	16
Understanding the VB Editor	19
Understanding Shortcomings of the Macro Recorder	21
Next Steps	32

With Lotus 1-2-3, you could record a macro today and play it back tomorrow, and it would faithfully work. When you attempt the same feat in Microsoft Excel, the macro might work today but not tomorrow. In 1995, when I tried to record my first Excel macro, I was horribly frustrated by this. In this book, I teach you the three rules for getting the most out of the macro recorder.

No One on the Excel Team Is Focused on the Macro Recorder

As Microsoft adds new features to Excel, the individual project manager for a feature makes sure that the macro recorder will record something when you execute the command. In the past decade, the recorded code might work in some situations, but it often does not work in all situations. If Microsoft had someone who was focused on creating a useful macro recorder, the recorded code could often be a lot more general than it currently is.

I once asked the project managers if they had a mission statement for the macro recorder. I asked them, “Are you trying to record code that will actually work or just trying to reveal the objects and methods so the person recording the code has to do more research to figure out how to use the commands?” The responses made me believe that no one at Microsoft actually cares about the macro recorder.

It used to be that you could record a command in any of five ways and the recorded code would work. Unfortunately, today, if you want to use the macro recorder, you often have to try recording the macro several different ways, until you find a set of steps that records code that reliably works.

Visual Basic Is Not Like BASIC

Two decades ago, the code generated by the macro recorder was unlike anything I had ever seen. It said this was “Visual Basic” (VB). I have had the pleasure of learning half a dozen programming languages at various times; this bizarre-looking language was horribly unintuitive and did not resemble the BASIC language I had learned in high school.

To make matters worse, even in 1995 I was the spreadsheet wizard in my office. My company had forced everyone to convert from Lotus 1-2-3 to Excel, which meant I was faced with a macro recorder that didn’t work and a language that I couldn’t understand. This was not a good combination of events.

My assumption in writing this book is that you are pretty talented with a spreadsheet. You probably know more than 90% of the people in your office. I also assume that even though you are not a programmer, you might have taken a class in BASIC at some point. However, knowing BASIC is not a requirement—it actually is a barrier to entry into the ranks of being a successful VBA programmer. There is a good chance that you have recorded a macro in Excel, and there’s a similar chance that you were not happy with the results.

Good News: Climbing the Learning Curve Is Easy

Even if you've been frustrated with the macro recorder, it is really just a small speed bump on your road to writing powerful programs in Excel. This book teaches you not only why the macro recorder fails but also how to change the recorded code into something useful. For all the former BASIC programmers in the audience, I decode VBA so that you can easily pick through recorded macro code and understand what is happening.

Great News: Excel with VBA Is Worth the Effort

Although you probably have been frustrated with Microsoft over the inability to record macros in Excel, the great news is that Excel VBA is powerful. Absolutely anything you can do in the Excel interface can be duplicated with stunning speed in Excel VBA. If you find yourself routinely creating the same reports manually day after day or week after week, Excel VBA will greatly streamline those tasks.

The authors of this book work for MrExcel Consulting. In this role, we have automated reports for hundreds of clients. The stories are often similar: The IT department has a several-month backlog of requests. Someone in accounting or engineering discovers that he or she can import some data into Excel and get the reports necessary to run the business. This is a liberating event: You no longer need to wait months for the IT department to write a program. However, the problem is that after you import the data into Excel and win accolades from your manager for producing the report, you will likely be asked to produce the same report every month or every week. This becomes very tedious.

Again, the great news is that with a few hours of VBA programming, you can automate the reporting process and turn it into a few button clicks. The reward is great. So hang with me as we cover a few of the basics.

This chapter exposes why the macro recorder does not work. It also walks through an example of recorded code and demonstrates why it works today but will fail tomorrow. I realize that the code you see in this chapter might not be familiar to you, but that's okay. The point of this chapter is to demonstrate the fundamental problem with the macro recorder. You'll also learn the fundamentals of the Visual Basic environment.

Knowing Your Tools: The Developer Tab

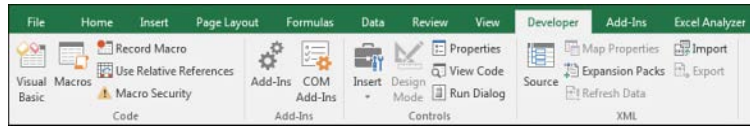
Let's start with a basic overview of the tools needed to use VBA. By default, Microsoft hides the VBA tools. You need to complete the following steps to change a setting to access the Developer tab:

1. Right-click the ribbon and choose Customize the Ribbon.
2. In the right list box, select the Developer check box, which is the eighth item.
3. Click OK to return to Excel.

Excel displays the Developer tab, as shown in Figure 1.1.

Figure 1.1

The Developer tab provides an interface for running and recording macros.



The Code group on the Developer tab contains the icons used for recording and playing back VBA macros, as listed here:

- **Visual Basic**—Opens the Visual Basic Editor.
- **Macros**—Displays the Macro dialog, where you can choose to run or edit a macro from the list of macros.
- **Record Macro**—Begins the process of recording a macro.
- **Use Relative References**—Toggles between using relative or absolute recording. With relative recording, Excel records that you move down three cells. With absolute recording, Excel records that you selected cell A4.
- **Macro Security**—Accesses the Trust Center, where you can choose to allow or disallow macros to run on this computer.

The Add-ins group provides icons for managing regular add-ins and COM add-ins.

The Controls group of the Developer tab contains an Insert menu where you can access a variety of programming controls that can be placed on the worksheet. See “Assigning a Macro to a Form Control, Text Box, or Shape,” later in this chapter. Other icons in this group enable you to work with the on-sheet controls. The Run Dialog button enables you to display a custom dialog box or userform that you designed in VBA. For more on userforms, see Chapter 10, “Userforms: An Introduction.”

The XML group of the Developer tab contains tools for importing and exporting XML documents.

The Modify group enables you to specify whether the Document Panel is always displayed for new documents. Users can enter keywords and a document description in the Document Panel. If you have SharePoint and InfoPath, you can define custom fields to appear in the Document Panel.

Understanding Which File Types Allow Macros

Excel 2016 offers support for four file types. Macros are not allowed to be stored in the .xlsx file type, and this file type is the default file type! You have to use the Save As setting for all of your macro workbooks, or you can change the default file type used by Excel 2016.

The available files types are as listed here:

- **Excel Workbook (.xlsx)**—Files are stored as a series of XML objects and then zipped into a single file. This creates significantly smaller file sizes. It also allows other applications (even Notepad!) to edit or create Excel workbooks. Unfortunately, macros cannot be stored in files with an .xlsx extension.
- **Excel Macro-Enabled Workbook (.xlsm)**—This is similar to the default .xlsx format, except macros are allowed. The basic concept is that if someone has an .xlsx file, he will not need to worry about malicious macros. However, if he sees an .xlsm file, he should be concerned that there might be macros attached.
- **Excel Binary Workbook (.xlsb)**—This is a binary format designed to handle the larger 1-million-row grid size introduced in Excel 2007. Legacy versions of Excel stored their files in a proprietary binary format. Although binary formats might load more quickly, they are more prone to corruption, and a few lost bits can destroy a whole file. Macros are allowed in this format.
- **Excel 97-2003 Workbook (.xls)**—This format produces files that can be read by anyone using legacy versions of Excel. Macros are allowed in this binary format; however, when you save in this format, you lose access to any cells outside A1:IV65536. In addition, if someone opens the file in Excel 2003, she loses access to anything that used features introduced in Excel 2007 or later.

To avoid having to choose a macro-enabled workbook in the Save As dialog, you can customize your copy of Excel to always save new files in the .xlsm format by following these steps:

1. Click the File menu and select Options.
2. In the Excel Options dialog, select the Save category from the left navigation pane.
3. Open the Save Files in This Format drop-down and select Excel Macro-Enabled Workbook (*.xlsm). Click OK.

NOTE

Although you and I are not afraid to use macros, I have encountered people who freak out when they see the .xlsm file type. They actually seem angry that I sent them an .xlsm file that did not have any macros. Their reaction seemed reminiscent of King Arthur's "You got me all worked up!" line in *Monty Python and the Holy Grail*. Google's Gmail has joined this camp, refusing to show a preview of any attachments sent in the .xlsm format.

If you encounter someone who seems to have a fear of the .xlsm file type, remind them of these points:

- Every workbook created in the past 30 years could have had macros, but in fact, most did not.
- If someone is trying to avoid macros, she should use the security settings to prevent macros from running anyway. The person can still open the .xlsm file to get the data in the spreadsheet.

With these arguments, I hope you can overcome any fears of the .xlsm file type so that it can be your default file type.

Macro Security

After a Word VBA macro was used as the delivery method for the Melissa virus, Microsoft changed the default security settings to prevent macros from running. Therefore, before we can begin discussing the recording of a macro, it's important to look at how to adjust the default settings.

In Excel 2016, you can either globally adjust the security settings or control macro settings for certain workbooks by saving the workbooks in a trusted location. Any workbook stored in a folder that is marked as a trusted location automatically has its macros enabled.

You can find the macro security settings under the Macro Security icon on the Developer tab. When you click this icon, the Macro Settings category of the Trust Center is displayed. You can use the left navigation bar in the dialog to access the Trusted Locations list.

Adding a Trusted Location

You can choose to store your macro workbooks in a folder that is marked as a trusted location. Any workbook stored in a trusted folder will have its macros enabled. Microsoft suggests that a trusted location should be on your hard drive. The default setting is that you cannot trust a location on a network drive.

To specify a trusted location, follow these steps:

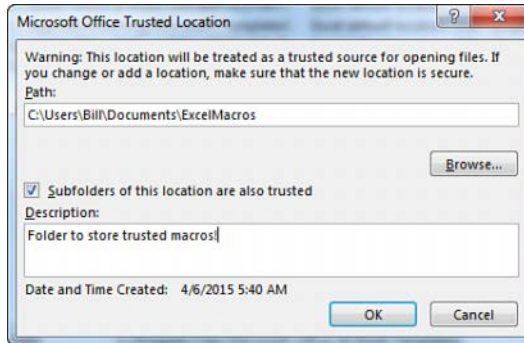
1. Click Macro Security in the Developer tab.
2. Click Trusted Locations in the left navigation pane of the Trust Center.
3. If you want to trust a location on a network drive, select Allow Trusted Locations on My Network.
4. Click the Add New Location button. Excel displays the Microsoft Office Trusted Location dialog (see Figure 1.2).
5. Click the Browse button. Excel displays the Browse dialog.
6. Browse to the parent folder of the folder you want to be a trusted location. Click the trusted folder. Although the folder name does not appear in the Folder Name box, click OK. The correct folder name will appear in the Browse dialog.
7. If you want to trust subfolders of the selected folder, select Subfolders of This Location Are Also Trusted.
8. Click OK to add the folder to the Trusted Locations list.

CAUTION

Use care when selecting a trusted location. When you double-click an Excel attachment in an email message, Outlook stores the file in a temporary folder on your C: drive. You will not want to globally add C:\ and all subfolders to the Trusted Locations list.

Figure 1.2

Manage trusted folders in the Trusted Locations category of the Trust Center.



Using Macro Settings to Enable Macros in Workbooks Outside Trusted Locations

For all macros not stored in a trusted location, Excel relies on the macro settings. The Low, Medium, High, and Very High settings that were familiar in Excel 2003 have been renamed.

To access the macro settings, click Macro Security in the Developer tab. Excel displays the Macro Settings category of the Trust Center dialog. Select the second option, Disable All Macros with Notification. A description of each option follows:

- **Disable All Macros Without Notification**—This setting prevents all macros from running. This setting is for people who never intend to run macros. Because you are currently holding a book that teaches you how to use macros, it is assumed that this setting is not for you. This setting is roughly equivalent to the old Very High security setting in Excel 2003. With this setting, only macros in the Trusted Locations folders can run.
- **Disable All Macros with Notification**—The operative words in this setting are “with Notification.” This means that you see a notification when you open a file with macros and you can choose to enable the content. If you ignore the notification, the macros remain disabled. This setting is similar to Medium security setting in Excel 2003 and is the recommended setting. In Excel 2016, a message is displayed in the Message Area indicating that macros have been disabled. You can choose to enable the content by clicking that option, as shown in Figure 1.3.
- **Disable All Macros Except Digitally Signed Macros**—This setting requires you to obtain a digital signing tool from VeriSign or another provider. This might be appropriate if you are going to be selling add-ins to others, but it’s a bit of a hassle if you just want to write macros for your own use.
- **Enable All Macros (Not Recommended: Potentially Dangerous Code Can Run)**—This setting is similar to the Low macro security setting in Excel 2003. Although it requires the least amount of hassle, it also opens your computer to attacks from malicious Melissa-like viruses. Microsoft suggests that you not use this setting.

Figure 1.3

The Enable Content option appears when you use Disable All Macros with Notification.



Using Disable All Macros with Notification

It is recommended that you set your macro settings to Disable All Macros with Notification. If you use this setting and open a workbook that contains macros, you see a security warning in the area just above the formula bar. If you are expecting macros in this workbook, click Enable Content. If you do not want to enable macros for the current workbook, dismiss the security warning by clicking the *X* at the far right of the message bar.

If you forget to enable the macros and attempt to run a macro, Excel indicates that you cannot run the macro because all macros have been disabled. If this occurs, close the workbook and reopen it to access the message bar again.

CAUTION

After you enable macros in a workbook stored on a local hard drive and then save the workbook, Excel remembers that you previously enabled macros in this workbook. The next time you open this workbook, macros are automatically enabled.

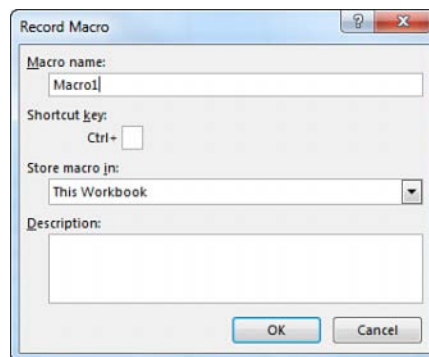
Overview of Recording, Storing, and Running a Macro

Recording a macro is useful when you do not have experience writing lines of code in a macro. As you gain more knowledge and experience, you will record macros less frequently.

To begin recording a macro, select Record Macro from the Developer tab. Before recording begins, Excel displays the Record Macro dialog box, as shown in Figure 1.4.

Figure 1.4

Use the Record Macro dialog box to assign a name and a shortcut key to the macro being recorded.



Filling Out the Record Macro Dialog

In the Macro Name field, type a name for the macro. Be sure to type continuous characters. For example, type **Macro1** without a space, not **Macro 1** with a space. Assuming that you will soon be creating many macros, use a meaningful name for the macro. A name such as FormatReport is more useful than one like Macro1.

The second field in the Record Macro dialog box is a shortcut key. If you type a lowercase j in this field and later press Ctrl+J, this macro runs. Be careful, however, because Ctrl+A through Ctrl+Z (except Ctrl+J) are all already assigned to other tasks in Excel. If you assign a macro to Ctrl+B, you won't be able to use Ctrl+B for bold anymore. One alternative is to assign the macros to Ctrl+Shift+A through Ctrl+Shift+Z. To assign a macro to Ctrl+Shift+A, you type Shift+A in the shortcut key box.

CAUTION

You can reuse a shortcut key for a macro. For example, if you assign a macro to Ctrl+C, Excel runs your macro instead of doing the normal action of copy.

In the Record Macro dialog box, choose where you want to save a macro when it is recorded: Personal Macro Workbook, New Workbook, or This Workbook. It is recommended that you store macros related to a particular workbook in This Workbook.

The Personal Macro Workbook (Personal.xlsm) is not a visible workbook; it is created if you choose to save the recording in the Personal Macro Workbook. This workbook is used to save a macro in a workbook that opens automatically when you start Excel, thereby enabling you to use the macro. After Excel is started, the workbook is hidden. If you want to display it, select Unhide from the View tab.

TIP

It is not recommended that you use the personal workbook for every macro you save. Save only those macros that assist you in general tasks—not in tasks that are performed in a specific sheet or workbook.

The fourth box in the Record Macro dialog is for a description. This description is added as a comment to the beginning of your macro.

After you select the location where you want to store the macro, click OK. Record your macro. For this example, type Hello World in the active cell and press Ctrl+Enter to accept the entry and stay in the same cell. When you are finished recording the macro, click the Stop Recording icon in the Developer tab.

TIP

You can also access a Stop Recording icon in the lower-left corner of the Excel window. Look for a small white square to the right of the word *Ready* in the status bar. Using this Stop button might be more convenient than returning to the Developer tab. After you record your first macro, this area usually has a Record Macro icon, which is a small dot on an Excel worksheet.

Running a Macro

If you assigned a shortcut key to your macro, you can play it by pressing the key combination. You can also assign macros to a button on the ribbon or the Quick Access Toolbar, form controls, or drawing objects, or you can run them from the Visual Basic toolbar.

Creating a Macro Button on the Ribbon

You can add an icon to a new group on the ribbon to run your macro. This is appropriate for macros stored in the Personal Macro Workbook. Icons added to the ribbon are still enabled even when your macro workbook is not open. If you click the icon when the macro workbook is not open, Excel opens the workbook and runs the macro. Follow these steps to add a macro button to the ribbon:

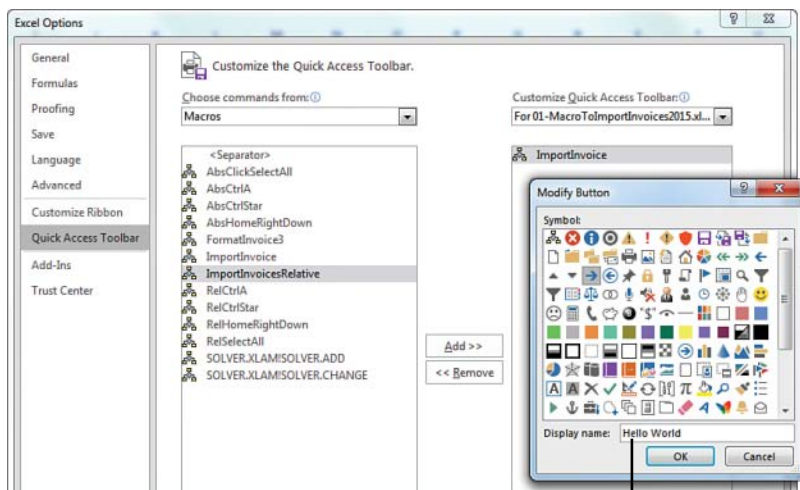
1. Right-click the ribbon and choose **Customize the Ribbon**.
2. In the list box on the right, choose the tab name where you want to add an icon.
3. Click the **New Group** button below the right list box. Excel adds a new entry called **New Group (Custom)** to the end of the groups in that ribbon tab.
4. To move the group to the left in the ribbon tab, click the up arrow icon on the right side of the dialog several times.
5. To rename the group, click the **Rename** button. Type a new name, such as **Report Macros**. Click **OK**. Excel shows the group in the list box as **Report Macros (Custom)**. Note that the word *Custom* does not appear in the ribbon.
6. Open the upper-left drop-down and choose **Macros** from the list. The **Macros** category is fourth in the list. Excel displays a list of available macros in the left list box.
7. Choose a macro from the left list box. Click the **Add** button in the center of the dialog. Excel moves the macro to the right list box in the selected group. Excel uses a generic VBA icon for all macros.
8. Click the macro in the right list box. Click the **Rename** button at the bottom of the right list box. Excel displays a list of 180 possible icons. Choose an icon. Alternatively, type a friendly label for the icon, such as **Format Report**.
9. You can move the **Report Macros** group to a new location on the ribbon tab. Click **Report Macros (Custom)** and use the up and down arrow icons on the right of the dialog.
10. Click **OK** to close the **Excel Options** dialog. The new button appears on the selected ribbon tab.

Creating a Macro Button on the Quick Access Toolbar

You can add an icon to the Quick Access Toolbar to run a macro. If a macro is stored in the Personal Macro Workbook, you can have the button permanently displayed in the Quick Access Toolbar. If the macro is stored in the current workbook, you can specify that the icon should appear only when the workbook is open. Follow these steps to add a macro button to the Quick Access Toolbar:

1. Right-click the Quick Access Toolbar and choose **Customize Quick Access Toolbar**.
2. If your macro should be available only when the current workbook is open, open the upper-right drop-down and change **For All Documents (Default)** to **For *FileName.xlsx***. Any icons associated with the current workbook are displayed at the end of the Quick Access Toolbar.
3. Open the upper-left drop-down and select **Macros** from the list. The **Macros** category is fourth in the list. Excel displays a list of available macros in the left list box.
4. Choose a macro from the left list box. Click the **Add** button in the center of the dialog. Excel moves the macro to the right list box. Excel uses a generic VBA icon for all macros.
5. Click the macro in the right list box. Click the **Modify** button at the bottom of the right list box. Excel displays a list of 180 possible icons (see Figure 1.5). Choose an icon from the list. In the **Display Name** box, replace the macro name with a short name that appears in the tooltip for the icon.
6. Click **OK** to close the **Modify Button** dialog.
7. Click **OK** to close the **Excel Options** dialog. The new button appears on the Quick Access Toolbar.

Figure 1.5
Attach a macro to a button on the Quick Access Toolbar.



Enter the ToolTip here

Assigning a Macro to a Form Control, Text Box, or Shape

If you want to create a macro specific to a workbook, you can store the macro in the workbook and attach it to a form control or any object on the sheet.

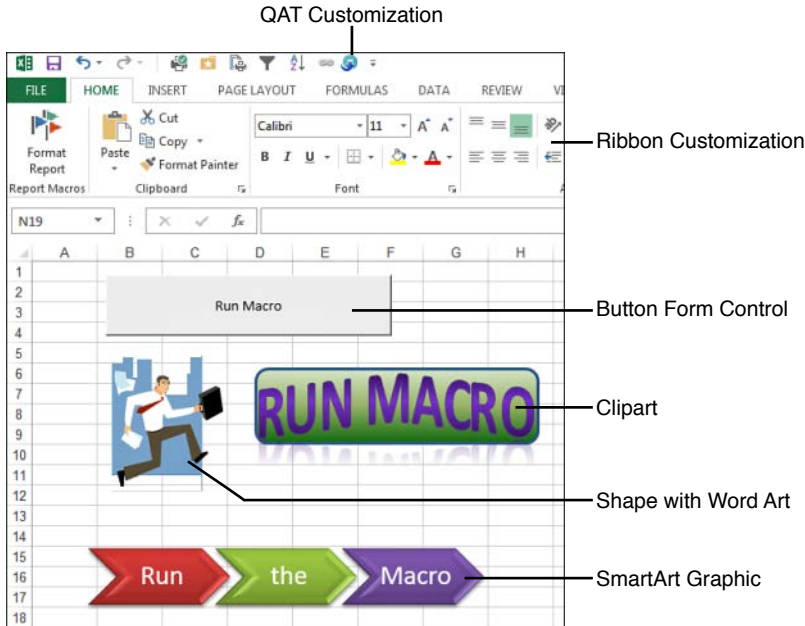
Follow these steps to attach a macro to a form control on the sheet:

1. On the Developer tab, click the Insert button to open its drop-down list. Excel offers 12 form controls and 12 ActiveX controls in this one drop-down menu. The form controls are at the top, and the ActiveX controls are at the bottom. Most icons in the ActiveX section of the drop-down look identical to an icon in the form controls section of the drop-down. Click the Button Form Control icon at the upper-left corner of the Insert drop-down.
2. Move your cursor over the worksheet; the cursor changes to a plus sign.
3. Draw a button on the sheet by clicking and holding the left mouse button while drawing a box shape. Release the button when you have finished.
4. Choose a macro from the Assign Macro dialog box and click OK. The button is created with generic text such as Button 1.
5. Type a new label for the button. Note that while you are typing, the selection border around the button changes from dots to diagonal lines to indicate that you are in Text Edit mode. You cannot change the button color while in Text Edit mode. To exit Text Edit mode, either click the diagonal lines to change them to dots or Ctrl+click the button again. Note that if you accidentally click away from the button, you should Ctrl+click the button to select it. Then drag the cursor over the text on the button to select the text.
6. Right-click the dots surrounding the button and select Format Control. Excel displays the Format Control dialog, which has seven tabs across the top. If your Format Control dialog has only a Font tab, you failed to exit Text Edit mode. If this occurred, close the dialog, Ctrl+click the button, and repeat this step.
7. Use the settings in the Format Control dialog to change the font size, font color, margins, and similar settings for the control. Click OK to close the Format Control dialog when you have finished. Click a cell to deselect the button.
8. Click the new button to run the macro.

Macros can be assigned to any worksheet object, such as clip art, a shape, SmartArt graphics, or a text box. In Figure 1.6, the top button is a traditional button form control. The other images are clip art, a shape with WordArt, and a SmartArt graphic. To assign a macro to any object, right-click the object and select Assign Macro.

Figure 1.6

Assigning a macro to a form control or an object is appropriate for macros stored in the same workbook as the control. You can assign a macro to any of these objects.



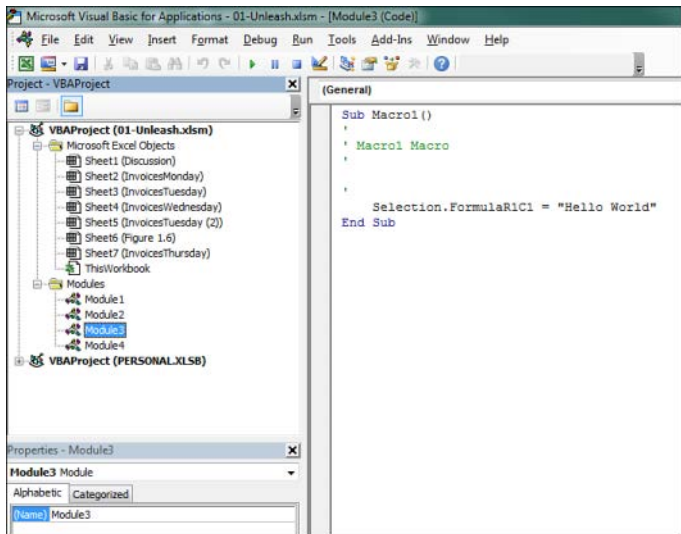
Understanding the VB Editor

If you want to edit a recorded macro, you do it in the VB Editor. Press Alt+F11 or use the Visual Basic icon in the Developer tab.

Figure 1.7 shows an example of a typical VB Editor screen. You can see three windows: the Project Explorer, the Properties window, and the Programming window. Don't worry if your window doesn't look exactly like this because you will see how to display the windows you need in this review of the editor.

Figure 1.7

The VB Editor window.



VB Editor Settings

Several settings in the VB Editor enable you to customize this editor and assist you in writing your macros.

Under Tools, Options, Editor, you find several useful settings. All settings except for one are set correctly by default. The remaining setting requires some consideration on your part. This setting is Require Variable Declaration. By default, Excel does not require you to declare variables. I prefer selecting this setting because it can save time when you create a program. My coauthor prefers to change this setting to require variable declaration. This change forces the compiler to stop if it finds a variable that it does not recognize, which reduces misspelled variable names. Whether you turn this setting on or keep it off is a matter of your personal preference.

The Project Explorer

The Project Explorer lists any open workbooks and add-ins that are loaded. If you click the + icon next to the VBA Project, you see that there is a folder containing Microsoft Excel objects. There can also be folders for forms, class modules, and standard modules. Each folder includes one or more individual components.

Right-clicking a component and selecting View Code or just double-clicking the component brings up any code in the Programming window. The exception is userforms, where double-clicking displays the userform in Design view.

To display the Project Explorer window, select View, Project Explorer from the menu or press Ctrl+R or locate the bizarre Project Explorer icon just below the Tools menu, sandwiched between Design Mode and Properties Window.

To insert a module, right-click your project, select Insert, and then choose the type of module you want. The available modules are as follows:

- **Microsoft Excel objects**—By default, a project consists of sheet modules for each sheet in the workbook and a single ThisWorkbook module. Code specific to a sheet such as controls or sheet events is placed on the corresponding sheet. Workbook events are placed in the ThisWorkbook module. You read more about events in Chapter 7, “Event Programming.”
- **Forms**—Excel enables you to design your own forms to interact with the user. You’ll read more about these forms in Chapter 10.
- **Modules**—When you record a macro, Excel automatically creates a module in which to place the code. Most of your code resides in these types of modules.
- **Class modules**—Class modules are Excel’s way of letting you create your own objects. They also allow pieces of code to be shared among programmers without the programmer’s needing to understand how it works. You read more about class modules in Chapter 9, “Creating Classes and Collections.”

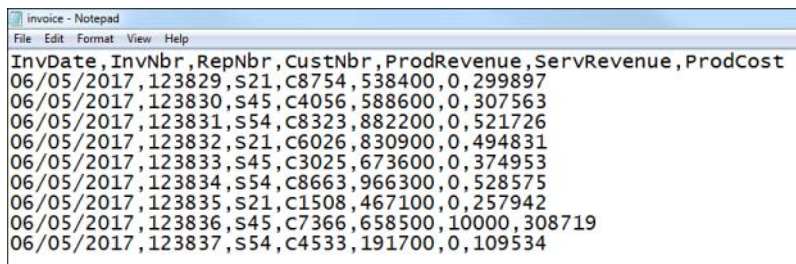
The Properties Window

The Properties window enables you to edit the properties of various components such as sheets, workbooks, modules, and form controls. The properties list varies according to what component is selected. To display this window, select View, Properties Window from the menu, press F4, or click the Project Properties icon on the toolbar.

Understanding Shortcomings of the Macro Recorder

Suppose you work in an accounting department. Each day you receive a text file from the company system showing all the invoices produced the prior day. This text file has commas separating the fields. The columns in the file are Invoice Date, Invoice Number, Sales Rep Number, Customer Number, Product Revenue, Service Revenue, and Product Cost (see Figure 1.8).

Figure 1.8
The Invoice.txt file.



```

invoice - Notepad
File Edit Format View Help
InvDate,InvNbr,RepNbr,CustNbr,ProdRevenue,ServRevenue,ProdCost
06/05/2017,123829,S21,C8754,538400,0,299897
06/05/2017,123830,S45,C4056,588600,0,307563
06/05/2017,123831,S54,C8323,882200,0,521726
06/05/2017,123832,S21,C6026,830900,0,494831
06/05/2017,123833,S45,C3025,673600,0,374953
06/05/2017,123834,S54,C8663,966300,0,528575
06/05/2017,123835,S21,C1508,467100,0,257942
06/05/2017,123836,S45,C7366,658500,10000,308719
06/05/2017,123837,S54,C4533,191700,0,109534
  
```

Each morning, you manually import this file into Excel. You add a total row to the data, bold the headings, and then print the report for distribution to a few managers.

This seems like a simple process that would be ideally suited to using the macro recorder. However, due to some problems with the macro recorder, your first few attempts might not be successful. The following case study explains how to overcome these problems.

CASE STUDY: PREPARING TO RECORD A MACRO

The task mentioned in the preceding section is perfect for a macro. However, before you record a macro, think about the steps you will use. In this case, the steps are as follows:

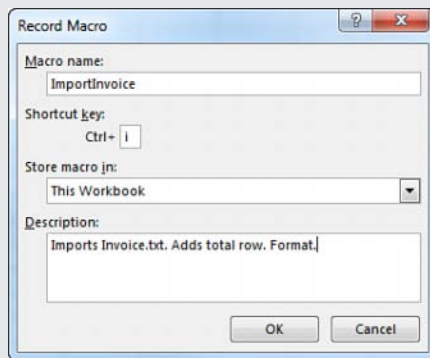
1. Click the File menu and select Open.
2. Navigate to the folder where Invoice.txt is stored.
3. Select All Files (*.*) from the Files of Type drop-down list.
4. Select Invoice.txt.
5. Click Open.
6. In the Text Import Wizard—Step 1 of 3 dialog, select Delimited from the Original Data Type section.
7. Click Next.
8. In the Text Import Wizard—Step 2 of 3 dialog, clear the Tab key and select Comma in the Delimiters section.

9. Click Next.
10. In the Text Import Wizard—Step 3 of 3 dialog, select General in the Column Data Format section and change it to Date: MDY.
11. Click Finish to import the file.
12. Press the Ctrl key and the down arrow key to move to the last row of data.
13. Press the down arrow one more time to move to the total row.
14. Type the word **Total**.
15. Press the right arrow key four times to move to column E of the total row.
16. Click the AutoSum button and press Ctrl+Enter to add a total to the Product Revenue column while remaining in that cell.
17. Click the AutoFill handle and drag it from column E to column G to copy the total formula to columns F and G.
18. Highlight row 1 and click the Bold icon on the Home tab to set the headings in bold.
19. Highlight the total row and click the Bold icon on the Home tab to set the totals in bold.
20. Press Ctrl+* to select the current region.
21. From the Home tab, select Format, AutoFit Column Width.

After you have rehearsed these steps in your head, you are ready to record your first macro. Open a blank workbook and save it with a name such as MacroToImportInvoices.xlsm. Click the Record Macro button on the Developer tab.

In the Record Macro dialog, the default macro name is Macro1. Change this to something descriptive like ImportInvoice. Make sure that the macros will be stored in This Workbook. You might want an easy way to run this macro later, so enter the letter i in the Shortcut Key field. In the Description field, add a little descriptive text to tell what the macro is doing (see Figure 1.9). Click OK when you are ready.

Figure 1.9
Before recording the macro, complete the Record Macro dialog box.



Recording the Macro

The macro recorder is now recording your every move. For this reason, perform your steps in exact order without extraneous actions. If you accidentally move to column F, type a value, clear the value, and then move back to E to enter the first total, the recorded macro will blindly make that same mistake day after day after day. Recorded macros move fast, but there is nothing like watching the macro recorder play out your mistakes repeatedly.

Carefully execute all the actions necessary to produce the report. After you have performed the final step, click the Stop Recording button in the Developer tab of the ribbon.

Examining Code in the Programming Window

Let's look at the code you just recorded from the case study. Don't worry if it doesn't make sense yet.

To open the VB Editor, press Alt+F11. In your VBA project (MacroToImportInvoices.xlsm), find the component Module1, right-click the module, and select View Code. Notice that some lines start with an apostrophe; these are comments and are ignored by the program. The macro recorder starts your macros with a few comments, using the description you entered in the Record Macro dialog. The comment for the keyboard shortcut is there to remind you of the shortcut.

NOTE

The comment does *not* assign the shortcut. If you change the comment to be Ctrl+J, it does not change the shortcut. You must change the setting in the Macro dialog box in Excel or run this line of code:

```
Application.MacroOptions Macro="ImportInvoice", _  
    Description="", ShortcutKey="j"
```

Recorded macro code is usually pretty neat (see Figure 1.10). Each noncomment line of code is indented 4 characters. If a line is longer than 100 characters, the recorder breaks it into multiple lines and indents the lines an additional 4 characters. To continue a line of code, type a space and an underscore at the end of the first line and then continue the code on the next line. Don't forget the space before the underscore. Using an underscore without the preceding space causes an error.

NOTE

The physical limitations of this book do not allow 100 characters on a single line. Therefore, the lines are broken at 80 characters so that they fit on a page. For this reason, your recorded macro might look slightly different from the ones that appear in this book.

Figure 1.10

The recorded macro is neat looking and nicely indented.

```
Sub ImportInvoice()
'
' ImportInvoice Macro
' Import Invoice.txt. Add Total Row. Format.
'
' Keyboard Shortcut: Ctrl+I
'
    Workbooks.OpenText Filename:="G:\2016VBA\SampleFiles\invoice.txt", Origin:= _
    437, StartRow:=1, DataType:=xlDelimited, TextQualifier:=xlDoubleQuote, _
    ConsecutiveDelimiter:=False, Tab:=False, Semicolon:=False, Comma:=True, _
    , Space:=False, Other:=False, FieldInfo:=Array(Array(1, 3), Array(2, 1), _
    Array(3, 1), Array(4, 1), Array(5, 1), Array(6, 1), Array(7, 1)), TrailingMinusNumbers _
    :=True
    Selection.End(xlDown).Select
    Range("A1").Select
    ActiveCell.FormulaR1C1 = "Total"
    Range("E11").Select
    Selection.FormulaR1C1 = "=SUM(R[-9]C:R[-1]C)"
    Selection.AutoFill Destination:=Range("E11:G11"), Type:=xlFillDefault
    Range("E11:G11").Select
    Rows("1:11").Select
    Selection.Font.Bold = True
    Rows("11:11").Select
    Selection.Font.Bold = True
    Selection.CurrentRegion.Select
    Selection.Columns.AutoFit
End Sub
```

Consider that the following seven lines of recorded code are actually only one line of code that has been broken into seven lines for readability:

```
Workbooks.OpenText Filename:="C:\somepath\invoice.txt", _
    Origin:=437, StartRow:=1, DataType:=xlDelimited, _
    TextQualifier:=xlDoubleQuote, ConsecutiveDelimiter:=False, _
    Tab:=True, Semicolon:=False, Comma:=True, Space:=False, _
    Other:=False, FieldInfo:=Array(Array(1, 3), Array(2, 1), _
    Array(3, 1), Array(4, 1), Array(5, 1), Array(6, 1), _
    Array(7, 1)), TrailingMinusNumbers:=True
```

Counting this as one line, the macro recorder was able to record the 21-step process in 14 lines of code, which is pretty impressive.

NOTE

Each action you perform in the Excel user interface might equate to one or more lines of recorded code. Some actions might generate a dozen lines of code.

Test Each Macro

It is always a good idea to test macros. To test your new macro, return to the regular Excel interface by pressing Alt+F11. Close Invoice.txt without saving any changes. MacroToImportInvoices.xls is still open.

Press Ctrl+I to run the recorded macro. It should work beautifully if you completed the steps correctly. The data is imported, totals are added, bold formatting is applied, and the columns are made wider. This seems like a perfect solution (see Figure 1.11).

Figure 1.11

The macro formats the data in the sheet.

	A	B	C	D	E	F	G
1	InvDate	InvNbr	RepNbr	CustNbr	ProdRevenue	ServRevenue	ProdCost
2	6/5/2017	123829	S21	C8754	538400	0	299897
3	6/5/2017	123830	S45	C4056	588600	0	307563
4	6/5/2017	123831	S54	C8323	882200	0	521726
5	6/5/2017	123832	S21	C6026	830900	0	494831
6	6/5/2017	123833	S45	C3025	673600	0	374953
7	6/5/2017	123834	S54	C8663	966300	0	528575
8	6/5/2017	123835	S21	C1508	467100	0	257942
9	6/5/2017	123836	S45	C7366	658500	10000	308719
10	6/5/2017	123837	S54	C4533	191700	0	109534
11	Total				5797300	10000	3203740

Running the Macro on Another Day Produces Undesired Results

After testing the macro, be sure to save your macro file to use on another day. But suppose that the next day, after receiving a new Invoice.txt file from the system, you open the macro and press Ctrl+I to run it, and disaster strikes. The data for June 5 happened to have 9 invoices, but the data for June 6 now has 17 invoices. The recorded macro blindly added the totals in Row 11 because this was where you put the totals when the macro was recorded (see Figure 1.12).

For those of you working along using the sample files in this book, follow these steps to try importing data for another day:

1. Close Invoice.txt in Excel.
2. In Windows Explorer, rename Invoice.txt to be Invoice1.txt.
3. In Windows Explorer, rename Invoice2.txt to be Invoice.txt.
4. Return to Excel and the MacroToImportInvoices.xlsm workbook.
5. Press Ctrl+I to run the macro with the larger data set.

This problem arises because the macro recorder is recording all your actions in Absolute mode by default. As an alternative to using the default state of the macro recorder, the next section discusses relative recording and how it might get you closer to the desired solution.

Figure 1.12

The intent of the recorded macro was to add a total at the end of the data, but the recorder made a macro that always adds totals at row 11.

	A	B	C	D	E	F	G
1	InvDate	InvNbr	RepNbr	CustNbr	ProdRevenue	ServRevenue	ProdCost
2	6/5/2017	123813	S82	C8754	716100	12000	423986
3	6/5/2017	123814		C4894	224200	0	131243
4	6/5/2017	123815	S43	C7278	277000	0	139208
5	6/5/2017	123816	S54	C6425	746100	15000	350683
6	6/5/2017	123817	S43	C6291	928300	0	488988
7	6/5/2017	123818	S43	C1000	723200	0	383069
8	6/5/2017	123819	S82	C6025	982600	0	544025
9	6/5/2017	123820	S17	C8026	490100	45000	243808
10	6/5/2017	123821	S43	C4244	615800	0	300579
11	Total	123822	S45	C1007	5703400	72000	3005589
12	6/5/2017	123823	S87	C1878	338100	0	165666
13	6/5/2017	123824	S43	C3068	567900	0	265775
14	6/5/2017	123825	S43	C7571	123456	0	55555
15	6/5/2017	123826	S55	C7181	37900	0	19811
16	6/5/2017	123827	S43	C7570	582700	0	292000
17	6/5/2017	123828	S87	C5302	495000	0	241504
18	6/5/2017	123828	S87	C5302	495000	0	241504

Possible Solution: Use Relative References When Recording

By default, the macro recorder records all actions as *absolute* actions. If you navigate to row 11 when you record the macro, the macro will always go to row 11 when the macro is run. This is rarely appropriate when dealing with variable numbers of rows of data. The better option is to use relative references when recording.

Macros recorded with absolute references note the actual address of the cell pointer, such as A11. Macros recorded with relative references note that the cell pointer should move a certain number of rows and columns from its current position. For example, if the cell pointer starts in cell A1, the code `ActiveCell.Offset(16, 1).Select` would move the cell pointer to B17, which is the cell 16 rows down and 1 column to the right.

Although relative recording is appropriate in most situations, there are times when you need to do something absolute while recording a macro. Here's a great example: After adding the totals to a data set, you need to return to row 1. If you simply click row 1 while in Relative mode, Excel records that you want to select the row 10 rows above the current row. This works with the first invoice file but not with longer or shorter invoice files. Here are two workarounds:

- Toggle relative recording off, click row 1, and then toggle relative recording back on.
- Keep relative recording turned on. Display the Go To dialog by pressing F5. Type **A1** and click OK. The Go To dialog gets recorded as always, going to the absolute address you typed, even if relative recording is turned on. A variation of this method is used in the following case study.

In the next case study, let's try the same task as before, this time using relative references. The solution will be much closer to working correctly.

CASE STUDY: RECORDING A MACRO WITH RELATIVE REFERENCES

Let's try to record the macro again, this time using relative references.

Note: If you are following along with the sample files, complete these steps:

1. Close Invoice.txt in Excel.
2. Rename Invoice.txt as Invoice2.txt.
3. Rename Invoice1.txt as Invoice.txt.
4. Return to the MacroToImportInvoices.xlsm workbook.

In the Developer tab, choose Use Relative References to toggle on relative recording. This setting persists until you turn it off or until you close Excel.

In the workbook MacroToImportInvoices.xlsm, record a new macro by selecting Record Macro from the Developer tab. Give the new macro the name ImportInvoicesRelative and assign a different shortcut key, such as Ctrl+J.

Repeat steps 1 through 11 in the previous case study to import the file and then follow these steps:

1. Press Ctrl+down arrow to move to the last row of data.
2. Press the down arrow key one more time to move to the total row.
3. Type the word **Total**.
4. Press the right arrow key four times to move to column E of the total row.
5. Hold the Shift key while pressing the right arrow key twice to select E11:G11.
6. Click the AutoSum button.
7. Press Shift+spacebar to select the entire row. Type Ctrl+B to apply bold formatting to it.
8. Press F5 to display the Go To dialog.
9. In the Go To dialog, type A1:G1 and click OK. Even though relative recording is turned on, any navigation through the Go To dialog box is recorded as an absolute reference. Press Ctrl+Home to move to cell A1.
10. Click the Bold icon to set the headings in bold.
11. Press Ctrl+* to select all data in the current region.
12. From the Home tab, select Format, AutoFit Column Width.
13. Stop recording.

Press Alt+F11 to go to the VB Editor to review your code. The new macro appears in Module1, below the previous macro.

If you close Excel between recording the first and second macros, Excel inserts a new module called Module2 for the newly recorded macro.

```
Sub ImportInvoicesRelative()  
    ' ImportInvoicesRelative Macro  
    ' Import. Total Row. Format.  
    ' Keyboard Shortcut: Ctrl+J
```

```

Workbooks.OpenText Filename:="C:\data\invoice.txt", _
    Origin:= 437, StartRow:=1, DataType:=xlDelimited, _
    TextQualifier:=xlDoubleQuote, ConsecutiveDelimiter:=False, _
    Tab:=False, Semicolon:=False, Comma:=True, Space:=False, _
    Other:=False, FieldInfo:=Array(Array(1, 3), Array(2, 1), _
    Array(3, 1), Array(4, 1), Array(5, 1), Array(6, 1), _
    Array(7, 1)), TrailingMinusNumbers:=True
Selection.End(xlDown).Select
ActiveCell.Offset(1, 0).Range("A1").Select
ActiveCell.FormulaR1C1 = "Total"
ActiveCell.Offset(0, 4).Range("A1:C1").Select
Selection.FormulaR1C1 = "=SUM(R[-9]C:R[-1]C)"
ActiveCell.Rows("1:1").EntireRow.Select
ActiveCell.Activate
Selection.Font.Bold = True
Application.Goto Reference:="R1C1:R1C7"
Selection.Font.Bold = True
Selection.CurrentRegion.Select
Selection.Columns.AutoFitSelection.Font.Bold = True
End Sub

```

To test the macro, close Invoice.txt without saving and then run the macro with Ctrl+J. Everything should look good, and you should get the same results as with the macro you created with the macro recorder.

The next test is to see whether the program works on the next day when you might have more rows. If you are working along with the sample files, close Invoice.txt in Excel. Rename Invoice.txt to Invoice1.txt. Rename Invoice2.txt to Invoice.txt.

Open MacroToImportInvoices.xls and run the new macro with Ctrl+J. This time, everything should look good, with the totals in the correct places. Look at Figure 1.13. Do you see anything out of the ordinary?

If you aren't careful, you might print these reports for your manager. If you did, you would be in trouble. When you look in cell E19, you can see that Excel has inserted a green triangle to tell you to look at the cell. If you happened to try this back in Excel 95 or Excel 97, before SmartTags, there would not have been an indicator that anything was wrong.

When you move the cell pointer to E19, an alert indicator pops up near the cell. This indicator tells you that the formula fails to include adjacent cells. If you look in the formula bar, you will see that the macro totaled only from row 10 to row 18. Neither the relative recording nor the nonrelative recording is smart enough to replicate the logic of the AutoSum button.

Imagine that you had fewer invoice records on this particular day. Excel would have rewarded you with the illogical formula =SUM(E6:E1048574), as shown in Figure 1.14. Since this formula would be in E7, circular reference warnings appear in the status bar.

Note: To try this yourself, close Invoice.txt in Excel. Rename Invoice.txt to Invoice2.txt. Rename Invoice4.txt to Invoice.txt.

Figure 1.13

The result of running the Relative macro.

	A	B	C	D	E	F	G
1	InvDate	InvNbr	RepNbr	CustNbr	ProdRevenue	ServRevenue	ProdCost
2	6/5/2017	123813	S82	C8754	716100	12000	423986
3	6/5/2017	123814		C4894	224200	0	131243
4	6/5/2017	123815	S43	C7278	277000	0	139208
5	6/5/2017	123816	S54	C6425	746100	15000	350683
6	6/5/2017	123817	S43	C6291	928300	0	488988
7	6/5/2017	123818	S43	C1000	723200	0	383069
8	6/5/2017	123819	S82	C6025	982600	0	544025
9	6/5/2017	123820	S17	C8026	490100	45000	243808
10	6/5/2017	123821	S43	C4244	615800	0	300579
11	6/5/2017	123822	S45	C1007	271300	0	153253
12	6/5/2017	123823	S87	C1878	338100	0	165666
13	6/5/2017	123824	S43	C3068	567900	0	265775
14	6/5/2017	123825	S43	C7571	123456	0	55555
15	6/5/2017	123826	S55	C7181	37900	0	19811
16	6/5/2017	123827	S43	C7570	582700	0	292000
17	6/5/2017	123828	S87	C5302	495000	0	241504
18	6/5/2017	123828	S87	C5302	495000	0	241504
19	Total				3527156	0	1735647

Figure 1.14

The result of running the Relative macro with fewer invoice records.

	A	B	C	D	E	F	G
1	InvDate	InvNbr	RepNbr	CustNbr	ProdRevenue	ServRevenue	ProdCost
2	6/8/2017	123850		C1654	161000	0	90761
3	6/8/2017	123851		C6460	275500	10000	146341
4	6/8/2017	123852		C5143	925400	0	473515
5	6/8/2017	123853		C7868	148200	0	75700
6	6/8/2017	123854		C3310	890200	0	468333
7	Total				0	0	0

If you have tried using the macro recorder, most likely you have run into problems similar to the ones produced in the previous two case studies. Although this is frustrating, you should be happy to know that the macro recorder actually gets you 95% of the way to a useful macro.

Your job is to recognize where the macro recorder is likely to fail and then be able to dive into the VBA code to fix the one or two lines that require adjusting to have a perfect macro. With some added human intelligence, you can produce awesome macros to speed up your daily work.

If you are like me, you are cursing Microsoft about now. We have wasted a good deal of time over a couple of days, and neither macro works. What makes it worse is that this sort of procedure would have been handled perfectly by the old Lotus 1-2-3 macro recorder introduced in 1983. Mitch Kapur solved this problem 33 years ago, and Microsoft still can't get it right.

Did you know that up through Excel 97, Microsoft Excel secretly ran Lotus command-line macros? I found this out right after Microsoft quit supporting Excel 97. At that time, a number of companies upgraded to Excel XP, which no longer supported the Lotus 1-2-3 macros. Many of these companies hired us to convert the old Lotus 1-2-3 macros to Excel VBA. It is interesting that in Excel 5, Excel 95, and Excel 97, Microsoft offered an

interpreter that could handle the Lotus macros that solved this problem correctly, yet its own macro recorder couldn't (and still can't!) solve the problem.

Never Use AutoSum or Quick Analysis While Recording a Macro

There actually is a macro recorder solution to the current problem with recording an AutoSum. It is important to recognize that the macro recorder will never correctly record the intent of the AutoSum button.

If you are in cell E99 and click the AutoSum button, Excel starts scanning from cell E98 upward until it locates a text cell, a blank cell, or a formula. It then proposes a formula that sums everything between the current cell and the found cell.

However, the macro recorder records the particular result of that search on the day that the macro was recorded. Rather than record something along the lines of "do the normal AutoSum logic," the macro recorder inserts a single line of code to add up the previous 98 cells.

Excel 2013 added the Quick Analysis feature. Select E2:G99; open Quick Analysis icon that appears below and to the right of a rectangular selection; choose Totals, Sum at Bottom; and you get the correct totals in row 100. The macro recorder hard-codes the formulas to always appear in row 100 and to always total row 2 through row 99.

The somewhat bizarre workaround is to type a SUM function that uses a mix of relative and absolute row references. If you type =SUM(E\$2:E10) while the macro recorder is running, Excel correctly adds code that always sums from a fixed row two down to the relative reference that is just above the current cell.

Here is the resulting code, with a few comments:

```
Sub FormatInvoice3()
Sub FormatInvoice3()
' FormatInvoice3 Macro
' Import. Total. Format.
' Keyboard Shortcut: Ctrl+K
Workbooks.OpenText Filename:="C:\Data\invoice.txt", _
    Origin:=437, StartRow:=1, DataType:=xlDelimited, _
    TextQualifier:=xlDoubleQuote, ConsecutiveDelimiter:=False, _
    Tab:=False, Semicolon:=False, Comma:=True, Space:=False, _
    Other:=False, FieldInfo:=Array(Array(1, 3), Array(2, 1), _
    Array(3, 1), Array(4, 1), Array(5, 1), Array(6, 1), _
    Array(7, 1)), TrailingMinusNumbers:=True
Selection.End(xlDown).Select
ActiveCell.Offset(1, 0).Range("A1").Select
ActiveCell.FormulaR1C1 = "Total"
ActiveCell.Offset(0, 4).Range("A1").Select
Selection.FormulaR1C1 = "=SUM(R2C:R[-1]C)"
Selection.AutoFill Destination:=ActiveCell.Range("A1:C1"), _
    Type:=xlFillDefault
ActiveCell.Range("A1:C1").Select
ActiveCell.Rows("1:1").EntireRow.Select
ActiveCell.Activate
Selection.Font.Bold = True
Application.Goto Reference:="R1C1:R1C7"
```

```
Selection.Font.Bold = True
Selection.CurrentRegion.Select
Selection.Columns.AutoFit
End Sub
```

This third macro consistently works with a data set of any size.

1

Four Tips for Using the Macro Recorder

You will rarely be able to record 100% of your macros and have them work. However, you will get much closer by using the four tips listed in the following subsections.

Tip 1: Turn on the Use Relative References Setting

Microsoft should have made this setting the default. Turn the setting on and leave it on while recording your macros.

Tip 2: Use Special Navigation Keys to Move to the Bottom of a Data Set

If you are at the top of a data set and need to move to the last cell that contains data, you can press Ctrl+down arrow or press the End key and then the down arrow key.

Similarly, to move to the last column in the current row of the data set, press Ctrl+right arrow or press End and then press the right arrow key.

By using these navigation keys, you can jump to the end of the data set, no matter how many rows or columns you have today.

Use Ctrl+* to select the current region around the active cell. Provided that you have no blank rows or blank columns in your data, this key combination selects the entire data set.

Tip 3: Never Touch the AutoSum Icon While Recording a Macro

The macro recorder does not record the “essence” of the AutoSum button. Instead, it hard-codes the formula that resulted from pressing the AutoSum button. This formula does not work any time you have more or fewer records in the data set.

Instead, type a formula with a single dollar sign, such as =SUM(E\$2:E10). When this is done, the macro recorder records the first E\$2 as a fixed reference and starts the SUM range directly below the row 1 headings. Provided that the active cell is E11, the macro recorder recognizes E10 as a relative reference pointing directly above the current cell.

Tip 4: Try Recording Different Methods if One Method Does Not Work

There are often many ways to perform tasks in Excel. If you encounter buggy code from one method, try another method. With 16 different project managers on the Excel team, it is likely that each method was programmed by a different group. In one of the case studies in this chapter, one task involved applying AutoFit Column Width to all cells. Some people might press Ctrl+A to select all cells. Others might press Ctrl+*. Since Excel 2007, the code generated by Ctrl+A when pressed in Relative mode does not work. The Ctrl+* code is very old and continues to work in all cases.

Next Steps

Chapter 2, “This Sounds Like BASIC, So Why Doesn’t It Look Familiar?” examines the three macros you recorded in this chapter to make more sense out of them. When you know how to decode the VBA code, it will feel natural to either correct the recorded code or simply write code from scratch. Hang on through one more chapter. You’ll soon learn that VBA is the solution, and you’ll be writing useful code that works consistently.

This Sounds Like BASIC, So Why Doesn't It Look Familiar?

2

I Can't Understand This Code

As mentioned in Chapter 1, “Unleashing the Power of Excel with VBA,” if you have taken a class in a procedural language such as BASIC or COBOL, you might be confused when you look at VBA code. Even though VBA stands for *Visual Basic for Applications*, it is an *object-oriented* version of BASIC. Here is a bit of VBA code:

```
Selection.End(xlDown).Select
Range("A11").Select
ActiveCell.FormulaR1C1 = "Total"
Range("E11").Select
Selection.FormulaR1C1 = _
    "=SUM(R[-9]C:R[-1]C)"
Selection.AutoFill _
    Destination:=Range("E11:G11"), _
    Type:=xlFillDefault
```

This code likely makes no sense to anyone who knows only procedural languages. Unfortunately, your first introduction to programming in school (assuming that you are over 40 years old) would have been a procedural language.

Here is a section of code written in the BASIC language:

```
For x = 1 to 10
    Print Rpt$(" ",x);
    Print "*"
Next x
```

IN THIS CHAPTER

I Can't Understand This Code	33
Understanding the Parts of VBA “Speech”	34
VBA Is Not Really Hard.....	37
Examining Recorded Macro Code: Using the VB Editor and Help	39
Using Debugging Tools to Figure Out Recorded Code	43
Object Browser: The Ultimate Reference	50
Seven Tips for Cleaning Up Recorded Code	51
Next Steps	57



If you run this code, you get a pyramid of asterisks on your screen:

```

*
 *
  *
   *
    *
   *
  *
 *
*

```

If you have ever been in a procedural programming class, you can probably look at the code and figure out what is going on because procedural languages are more English-like than object-oriented languages. The statement `Print "Hello World"` follows the verb-object format, which is how you would generally talk. Let's step away from programming for a second and look at a concrete example.

Understanding the Parts of VBA "Speech"

If you were going to write code for instructions to play soccer using BASIC, the instruction to kick a ball would look something like this:

```
"Kick the Ball"
```

Hey, this is how you talk! It makes sense. You have a verb (*kick*) and then a noun (*ball*). The BASIC code in the preceding section has a verb (`Print`) and a noun (the asterisk, `*`). Life is good.

Here is the problem: VBA doesn't work like this. In fact, no object-oriented language works like this. In an object-oriented language, the objects (nouns) are most important, hence the name: object-oriented. If you were going to write code for instructions to play soccer with VBA, the basic structure would be as follows:

```
Ball.Kick
```

You have a noun (`Ball`), which comes first. In VBA, this is an *object*. Then you have the verb (`Kick`), which comes next. In VBA, this is a *method*.

The basic structure of VBA is a bunch of lines of code with this syntax:

```
Object.Method
```

Needless to say, this is not English. If you took a romance language in high school, you will remember that those languages use a "noun-adjective" construct. However, no one uses "noun-verb" to tell someone to do something:

```

Water.Drink
Food.Eat
Girl.Kiss

```

That is why VBA is confusing to someone who previously took a procedural programming class.

Let's carry the analogy a bit further. Imagine that you walk onto a grassy field, and there are five balls in front of you. There are a soccer ball, basketball, baseball, bowling ball, and tennis ball. You want to instruct a kid on your soccer team to "kick the soccer ball."

If you tell him to kick the ball (or `ball.kick`), you really aren't sure which one of the five balls he will kick. Maybe he will kick the one closest to him, which could be a problem if he is standing in front of the bowling ball.

For almost any noun, or object in VBA, there is a collection of that object. Think about Excel. If you can have one row, you can have a bunch of rows. If you can have one cell, you can have a bunch of cells. If you can have one worksheet, you can have a bunch of worksheets. The only difference between an object and a collection is that you add an *s* to the name of the object:

```
Row becomes Rows.  
Cell becomes Cells.  
Ball becomes Balls.
```

When you refer to something that is a collection, you have to tell the programming language to which item you are referring. There are a couple of ways to do this. You can refer to an item by using a number. For example, if the soccer ball is the second ball, you might say this:

```
Balls(2).Kick
```

This works fine, but it could be a dangerous way to program. For example, it might work on Tuesday. However, if you get to the field on Wednesday and someone has rearranged the balls, `Balls(2).Kick` might be a painful exercise.

A much safer way to go is to use a name for the object in a collection. You can say the following:

```
Balls("Soccer").Kick
```

With this method, you always know that it will be the soccer ball that is being kicked.

So far, so good. You know that a ball will be kicked, and you know that it will be the soccer ball. For most of the verbs, or methods in Excel VBA, there are *parameters* that tell *how* to do the action. These parameters act as adverbs. You might want the soccer ball to be kicked to the left and with a hard force. In this case, the method would have a number of parameters that tell how the program should perform the method:

```
Balls("Soccer").Kick Direction:=Left, Force:=Hard
```

When you are looking at VBA code, the colon-equal sign combination (`:=`) indicates that you are looking at parameters of how the verb should be performed.

Sometimes, a method will have a list of 10 parameters, some of which are optional. For example, if the `Kick` method has an `Elevation` parameter, you would have this line of code:

```
Balls("Soccer").Kick Direction:=Left, Force:=Hard, Elevation:=High
```

Here is the confusing part: Every method has a default order for its parameters. If you are not a conscientious programmer, and you happen to know the order of the parameters, you can leave off the parameter names. The following code is equivalent to the previous line of code:

```
Balls("Soccer").Kick Left, Hard, High
```

This throws a monkey wrench into our understanding. Without `:=`, it is not obvious that you have parameters. Unless you know the parameter order, you might not understand what is being said. It is pretty easy with `Left`, `Hard`, and `High`, but when you have parameters like the following:

```
ActiveSheet.Shapes.AddShape type:=1, Left:=10, Top:=20, _  
    Width:=100, Height:=200
```

it gets confusing if you instead have this:

```
ActiveSheet.Shapes.AddShape 1, 10, 20, 100, 200
```

The preceding line is valid code. However, unless you know that the default order of the parameters for this `Add` method is `Type`, `Left`, `Top`, `Width`, `Height`, this code does not make sense. The default order for any particular method is the order of the parameters as shown in the `Help` topic for that method.

To make life more confusing, you are allowed to start specifying parameters in their default order without naming them, and then you can switch to naming parameters when you hit one that does not match the default order. If you want to kick the ball to the left and high but do not care about the force (that is, you are willing to accept the default force), the following two statements are equivalent:

```
Balls("Soccer").Kick Direction:=Left, Elevation:=High  
Balls("Soccer").Kick Left, Elevation:=High
```

However, keep in mind that as soon as you start naming parameters, they have to be named for the remainder of that line of code.

Some methods simply act on their own. To simulate pressing the `F9` key, you use this code:

```
Application.Calculate
```

Other methods perform an action and create something. For example, you can add a worksheet by using the following:

```
Worksheets.Add Before:=Worksheets(1)
```

However, because `Worksheets.Add` creates a new object, you can assign the results of this method to a variable. In this case, you must surround the parameters with parentheses:

```
Set MyWorksheet = Worksheets.Add(Before:=Worksheets(1))
```

One final bit of grammar is necessary: adjectives. Just as adjectives describe a noun, *properties* describe an object. Because you are an Excel fan, let's switch from the soccer analogy to an Excel analogy. There is an object to describe the active cell. Fortunately, it has a very intuitive name:

```
ActiveCell
```

Suppose you want to change the color of the active cell to red. There is a property called `Interior.Color` for a cell that uses a complex series of codes. However, you can turn a cell to red by using this code:

```
ActiveCell.Interior.Color = 255
```

You can see how this can be confusing. Again, there is the *noun-dot-something* construct, but this time it is `Object.Property` rather than `Object.Method`. How you tell them apart is quite subtle: There is no colon before the equal sign. A property is almost always set equal to something, or perhaps the value of a property is assigned to something else.

To make this cell color the same as cell A1, you might say this:

```
ActiveCell.Interior.Color = Range("A1").Interior.Color
```

`Interior.Color` is a property. By changing the value of a property, you can make things look different. It is kind of bizarre: Change an adjective, and you are actually doing something to the cell. Humans would say, “Color the cell red,” whereas VBA says this:

```
ActiveCell.Interior.Color = 255
```

Table 2.1 summarizes the VBA “parts of speech.”

Table 2.1 Parts of the VBA Programming Language

VBA Component	Analogous To	Notes
Object	Noun	Examples include cell or sheet.
Collection	Plural noun	Usually specifies which object: <code>Worksheets(1)</code> .
Method	Verb	Appears as <code>Object.Method</code> .
Parameter	Adverb	Lists parameters after the method. Separate the parameter name from its value with <code>:</code> .
Property	Adjective	You can set a property (for example, <code>activecell.height=10</code>) or store the value of a property (for example, <code>x = activecell.height</code>).

VBA Is Not Really Hard

Knowing whether you are dealing with properties or methods helps you set up the correct syntax for your code. Don’t worry if it all seems confusing right now. When you are writing VBA code from scratch, it is tough to know whether the process of changing a cell to yellow requires a verb or an adjective. Is it a method or a property?

This is where the macro recorder is especially helpful. When you don’t know how to code something, you record a short little macro, look at the recorded code, and figure out what is going on.

VBA Help Files: Using F1 to Find Anything

Excel VBA Help is an amazing feature, provided that you are connected to the Internet. If you are going to write VBA macros, you absolutely *must* have access to the VBA Help topics installed. Follow these steps to see how easy it is to get help in VBA:

1. Open Excel and switch to the VB Editor by pressing Alt+F11. From the Insert menu, select Module.
2. Type these three lines of code:

```
Sub Test()  
    MsgBox "Hello World!"  
End Sub
```
3. Click inside the word `MsgBox`.
4. With the cursor in the word `MsgBox`, press F1. If you can reach the Internet, you see the Help topic for the `MsgBox` function.

Using Help Topics

If you request help on a function or method, the Help topic walks you through the various available arguments. If you browse to the bottom of a Help topic, you can see a great resource: code samples under the Example heading (see Figure 2.1).

It is possible to select the code, copy it to the Clipboard by pressing Ctrl+C, and then paste it into a module by pressing Ctrl+V.

After you record a macro, if there are objects or methods about which you are unsure, you can get help by inserting the cursor in any keyword and pressing F1.

Figure 2.1
Most Help topics include
code samples.



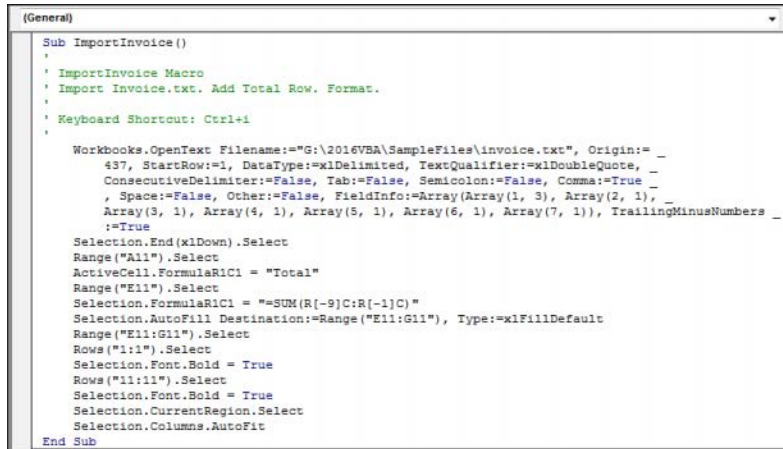
Examining Recorded Macro Code: Using the VB Editor and Help

Let's take a look at the code you recorded in Chapter 1 to see whether it makes more sense now that you know about objects, properties, and methods. You can also see whether it's possible to correct the errors created by the macro recorder.

Figure 2.2 shows the first code that Excel recorded in the example from Chapter 1.

Figure 2.2

Recorded code from the example in Chapter 1.



```

Sub ImportInvoice()
'
' ImportInvoice Macro
' Import Invoice.txt. Add Total Row. Format.
'
' Keyboard Shortcut: Ctrl+I
'
Workbooks.OpenText Filename:="G:\2016VBA\SampleFiles\Invoice.txt", Origin:= _
437, StartRow:=1, DataType:=xlDelimited, TextQualifier:=xlDoubleQuote, _
ConsecutiveDelimiter:=False, Tab:=False, Semicolon:=False, Comma:=True, _
Space:=False, Other:=False, FieldInfo:=Array(Array(1, 3), Array(2, 1), _
Array(3, 1), Array(4, 1), Array(5, 1), Array(6, 1), Array(7, 1)), TrailingMinusNumbers _
:=True
Selection.End(xlDown).Select
Range("All").Select
ActiveCell.FormulaR1C1 = "Total"
Range("E11").Select
Selection.FormulaR1C1 = "=SUM(R[-9]C:R[-1]C)"
Selection.AutoFill Destination:=Range("E11:G11"), Type:=xlFillDefault
Range("E11:G11").Select
Rows("1:1").Select
Selection.Font.Bold = True
Rows("11:11").Select
Selection.Font.Bold = True
Selection.CurrentRegion.Select
Selection.Columns.AutoFit
End Sub

```

Now that you understand the concept of `Noun.Verb` or `Object.Method`, consider the first line of code that says `Workbooks.OpenText`. In this case, `Workbooks` is an object, and `OpenText` is a method. Click your cursor inside the word `OpenText` and press F1 for an explanation of the `OpenText` method (see Figure 2.3).

The Help file confirms that `OpenText` is a method, or an action word. The default order for all the arguments that can be used with `OpenText` appears in the gray box. Notice that only one argument is required: `Filename`. All the other arguments are listed as optional.

Optional Parameters

The Help file can tell you if you happen to skip an optional parameter. For `StartRow`, the Help file indicates that the default value is 1. If you leave out the `StartRow` parameter, Excel starts importing at row 1. This is fairly safe.

Now look at the Help file note about `Origin`. If this argument is omitted, you inherit whatever value was used for `Origin` the last time someone used this feature in Excel on this computer. That is a recipe for disaster. For example, your code might work 98% of the time. However, immediately after someone imports an Arabic file, Excel remembers the setting for Arabic and thereafter assumes that this is what your macro wants if you don't explicitly code this parameter.

Figure 2.3
Part of the help topic for the `OpenText` method.

Parameters			
<i>Filename</i>	Required	String	Specifies the file name of the text file to be opened and parsed.
<i>Origin</i>	Optional	Variant	Specifies the origin of the text file. Can be one of the following <code>XlPlatform</code> constants: <code>xlMacintosh</code> , <code>xlWindows</code> , or <code>xlMSDOS</code> . Additionally, this could be an integer representing the code page number of the desired code page. For example, "1256" would specify that the encoding of the source text file is Arabic (Windows). If this argument is omitted, the method uses the current setting of the File Origin option in the Text Import Wizard .
<i>StartRow</i>	Optional	Variant	The row number at which to start parsing text. The default value is 1.
<i>DataType</i>	Optional	Variant	Specifies the column format of the data in the file. Can be one of the following <code>XlTextParsingType</code> constants: <code>xlDelimited</code> or <code>xlFixedWidth</code> . If this argument is not specified, Microsoft Excel attempts to determine the column format when it opens the file.
<i>TextQualifier</i>	Optional	XlTextQualifier	Specifies the text qualifier.
<i>ConsecutiveDelimiter</i>	Optional	Variant	True to have consecutive delimiters considered one delimiter. The default is False .

Defined Constants

Look at the Help file entry for `DataType` in Figure 2.3, which says it can be one of these constants: `xlDelimited` or `xlFixedWidth`. The Help file says these are the valid `xlTextParsingType` constants that are predefined in Excel VBA. In the VB Editor, press `Ctrl+G` to bring up the Immediate window. In the Immediate window, type this line and press Enter:

```
Print xlFixedWidth
```

The answer appears in the Immediate window. `xlFixedWidth` is the equivalent of saying 2 (see Figure 2.4). In the Immediate window type `Print xlDelimited`, which is really the same as typing 1. Microsoft correctly assumes that it is easier for someone to read code that uses the somewhat English-like term `xlDelimited` than to read 1.

Figure 2.4
In the Immediate window of the VB Editor, query to see the true value of constants such as `xlFixedWidth`.

Immediate
<code>Print xlFixedWidth</code>
2
<code>Print xlDelimited</code>
1

If you were an evil programmer, you could certainly memorize all these constants and write code using the numeric equivalents of the constants. However, the programming gods (and the next person who has to look at your code) will curse you for this.

In most cases, the Help file either specifically calls out the valid values of the constants or offers a hyperlink that opens the Help topic showing the complete enumeration and the valid values for the constants (see Figure 2.5).

One complaint with this excellent Help system is that it does not identify which parameters are new to a given version. In this particular case, `TrailingMinusNumbers` was introduced in Excel 2002. If you attempt to give this program to someone who is still using Excel 2000, the code does not run because Excel does not understand the `TrailingMinusNumbers` parameter. Sadly, the only way to learn to handle this frustrating problem is through trial and error.

If you read the Help topic on `OpenText`, you can surmise that it is basically the equivalent of opening a file using the Text Import Wizard. In step 1 of the wizard, you normally choose either Delimited or Fixed Width. You also specify the file origin and at which row to start. This first step of the wizard is handled by these parameters of the `OpenText` method:

```
Origin:=437
StartRow:=1
DataType:=xlDelimited
```

2

Figure 2.5

Click the hyperlink to see all the possible constant values. Here, the 10 possible `xlColumnDataType` constants are revealed in a new help topic.

xlColumnDataType Enumeration (Excel)		
Office 2013 Other Versions ▾		
Specifies how a column is to be parsed.		
Version Information		
Version Added: Excel 2007		
Name	Value	Description
<code>xlIDMYFormat</code>	4	DMY date format.
<code>xlIDYMFormat</code>	7	DYM date format.
<code>xlEMDFormat</code>	10	EMD date format.
<code>xlGeneralFormat</code>	1	General.
<code>xlMDYFormat</code>	3	MDY date format.
<code>xlIMYDFormat</code>	6	MYD date format.
<code>xlSkipColumn</code>	9	Column is not parsed.
<code>xlTextFormat</code>	2	Text.
<code>xlYDMFormat</code>	8	YDM date format.
<code>xlYMDFormat</code>	5	YMD date format.

Step 2 of the Text Import Wizard enables you to specify that your fields be delimited by commas. Because you do not want to treat two commas as a single comma, the `Treat Consecutive Delimiters as One` check box should not be selected. Sometimes, a field may contain a comma, such as “XYZ, Inc.” In this case, the field should have quotes around the value, as specified in the Text Qualifier box. This second step of the wizard is handled by the following parameters of the `OpenText` method:

```
TextQualifier:=xlDoubleQuote
ConsecutiveDelimiter:=False
```

```
Tab:=False  
Semicolon:=False  
Comma:=True  
Space:=False  
Other:=False
```

Step 3 of the wizard is where you actually identify the field types. In this case, you leave all fields as General except for the first field, which is marked as a date in MDY (Month, Day, Year) format. This is represented in code by the `FieldInfo` parameter.

The third step of the Text Import Wizard is fairly complex. The entire `FieldInfo` parameter of the `OpenText` method duplicates the choices made on this step of the wizard. If you happen to click the Advanced button on the third step of the wizard, you have an opportunity to specify something other than the default decimal and thousands separators, as well as the setting Trailing Minus for Negative Numbers.

TIP

Note that the macro recorder does not write code for `DecimalSeparator` or `ThousandsSeparator` unless you change these from the defaults. The macro recorder does, however, always record the `TrailingMinusNumbers` parameter.

Remember that every action you perform in Excel while recording a macro gets translated to VBA code. In the case of many dialog boxes, the settings you do not change are often recorded along with the items you do change. When you click OK to close the dialog, the macro recorder often records all the current settings from the dialog in the macro.

Here is another example. The next line of code in the macro is this:

```
Selection.End(xlDown).Select
```

You can click to get help for three topics in this line of code: `Selection`, `End`, and `Select`. Assuming that `Selection` and `Select` are somewhat self-explanatory, click in the word `End` and press F1 for Help. A Context Help dialog box appears, saying that there are two possible Help topics for `End`—one in the Excel library and one in the VBA library.

If you are new to VBA, you might not know which Help library to select. Select one and then click Help. In this case, the `End` Help topic in the VBA library is talking about the `End` statement, which is not what you need.

Close Help, press F1 again, and select the `End` object in the Excel library. This Help topic says that `End` is a property. It returns a `Range` object that is equivalent to pressing End+up arrow or End+down arrow in the Excel interface (see Figure 2.6). If you click the blue hyperlink for `xlDirection`, you see the valid parameters that can be passed to the `End` function.